CakePHP Cookbook Documentation

Versão 2.x

Cake Software Foundation

08 jul, 2019

Conteúdo

CAPÍTULO 1

Primeiros Passos

O framework CakePHP fornece uma base robusta e sólida para suas aplicações. Podendo tratar todos os aspectos, da requisição inicial do usuário até a renderização de uma página web. Visto que o framework segue o princípio MVC, ele lhe permite customizar e estender facilmente muitos dos aspectos de sua aplicação.

O framework também fornece uma organização estrutural básica de nome de arquivos a nomes de tabelas do banco de dados, mantendo toda sua aplicação consistente e lógica. Este conceito é simples mas poderoso. Siga as convenções e você sempre saberá exatamente onde as coisas estão e como estão organizadas.

A melhor maneira de experimentar e aprender o CakePHP é sentar em frente ao computador e construir alguma coisa. Para começar vamos construir um blog simples.

Blog

Bem vindo ao CakePHP. Você provavelmente está lendo este tutorial porque quer aprender mais sobre como o CakePHP funciona. Nosso objetivo é aumentar a produtividade e fazer a programação uma tarefa mais divertida: Esperamos que você veja isto na prática enquanto mergulha nos códigos.

Este tutorial irá cobrir a criação de uma aplicação de blog simples. Nós iremos baixar e instalar o Cake, criar e configurar o banco de dados e criar a lógica da aplicação suficiente para listar, adicionar, editar e deletar posts do blog.

Aqui vai uma lista do que você vai precisar:

1. Um servidor web rodando. Nós iremos assumir que você esteja usando o Apache, embora as instruções para usar outros servidores sejam bem semelhantes. Talvez tenhamos que brincar um pouco com as configurações do servidor mas a maioria das pessoas serão capazes de ter o Cake rodando sem precisar configurar nada.
2. Um servidor de banco de dados. Nós iremos usar o MySQL Server neste tutorial. Você precisa saber o mínimo sobre SQL para criar um banco de dados. O Cake pegará as rédeas a partir deste ponto.
3. Conhecimento básico da linguagem PHP. Quanto mais orientado a objetos você já programou, melhor: mas não tenha medo se é fã de programação procedural.

4. E por último, você vai precisar de um conhecimento básico do padrão de projetos MVC. Uma rápida visão geral pode ser encontrada em *Entendendo o Model-View-Controller*. Não se preocupe, deve ter meia página ou menos.

Então, vamos começar!

Baixando o Cake

Primeiro, vamos baixar uma cópia recente do CakePHP.

Para fazer o download de uma cópia recente, visite o projeto do CakePHP no github: https://github.com/cakephp/cakephp/downloads e faça o download da última versão 2.0.

Você também pode clonar o repositório usando o git[1]. `git clone git://github.com/cakephp/cakephp.git`.

Idependente da maneira de como você baixou o Cake, coloque o código obtido dentro do seu diretório web público. A estrutura dos diretórios deve ficar parecido com o seguinte:

```
/caminho_para_diretorio_web_publico
    /app
    /lib
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

Agora pode ser um bom momento para aprender um pouco sobre como funciona a estrutura de diretórios do CakePHP: Veja a seção *Estrutura de Diretórios no CakePHP*.

Criando o Banco de Dados do Blog

Em seguida, vamos configurar o banco de dados correspondente ao nosso blog. Se você já não tiver feito isto, crie um banco de dados vazio para usar neste tutorial com o nome que desejar. Neste momento, vamos criar apenas uma tabela para armazenar nossos posts. Também vamos inserir alguns posts para usar como teste. Execute as instruções a seguir no seu banco de dados:

```
-- Primeiro, criamos nossa tabela de posts
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

-- Agora inserimos alguns posts para testar
INSERT INTO posts (title, body, created)
    VALUES ('The title', 'This is the post body.', NOW());
INSERT INTO posts (title, body, created)
    VALUES ('A title once again', 'And the post body follows.', NOW());
INSERT INTO posts (title, body, created)
    VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

[1] http://git-scm.com/

A escolha do nome de tabelas e colunas não são arbitrárias. Se você seguir as convenções de nomenclatura para estruturas do banco de dados e as convenções para nomes de classes (ambas descritas em *Convenções no CakePHP*), você será capaz de tirar proveito de muitas funcionalidades do CakePHP e evitar arquivos de configurações. O Cake é flexivel o bastante para acomodar até mesmo os piores esquemas de banco de dados legados, mas aderindo as convenções você poupa seu tempo.

Veja *Convenções no CakePHP* para mais informações. Aqui, basta dizer que ao nomear nossa tabela de "posts", automaticamente ela será ligada ao nosso model Post e as colunas "modified" e "created" serão «automagicamente» atualizadas pelo CakePHP.

Configurações do Banco de Dados

Para o Alto e Avante: Vamos agora avisar ao Cake onde está nosso banco de dados e como conectar a ele. Para muitos, esta é a primeira e última configuração a ser feita.

Uma exemplo de arquivo de configuração do banco de dados pode ser encontrado em `/app/Config/database.php.default`. Copie este arquivo no mesmo diretório renomeando-o para `database.php`.

O arquivo é bem simples: basta alterar os valores da variável $default com os dados da nossa configuração. Um exemplo completo desta configuração irá se parecer com esta:

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

Após salvar seu novo arquivo database.php, você estará apto para abrir seu navegador e ver a página de boas vindas do Cake. A página de boas vindas deverá lhe mostrar uma mensagem dizendo que seu arquivo de conexão com o banco de dados foi encontrado, e que o Cake conseguiu se conectar com o banco de dados.

Configuração Opcional

Existem outros três itens que podem ser configurados. Muitos desenvolvedores sempre configuram estes itens, mas eles não são obrigatórios para este tutorial. Uma das configurações é customizar uma string (ou «salt») para ser utilizada nos hashes de segurança. O segundo é definir um número (ou «seed») para uso em criptografia. E o terceiro é dar permissão de escrita para o CakePHP na pasta `tmp`.

O «security salt» é utilizado para gerar hashes. Altere o valor padrão do salt editando o arquivo `/app/Config/core.php` na linha 187. Não importa muito o que o novo valor seja, basta que não seja fácil de adivinhar.

```
/**
 * A random string used in security hashing methods.
 */
Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
?>
```

O «cipher seed» é usado para criptografar/descriptografar strings. Altere o valor padrão editando o arquivo `/app/Config/core.php` na linha 192. Como no «security salt», não importa muito o que o novo valor seja, basta que não seja fácil de adivinhar.

```
/**
 * A random numeric string (digits only) used to encrypt/decrypt strings.
 */
Configure::write('Security.cipherSeed', '7485712659625147843639846751');
?>
```

A última tarefa é garantir acesso de escrita para a pasta `app/tmp`. A melhor maneira para fazer isto é localizar o usuário com que o seu servidor web é executado (`<?php echo `whoami`; ?>`) e alterar o dono da pasta `app/tmp` para este usuário. Você pode executar (em *nix) o comando a seguir para alterar o usuário dono da pasta.

```
$ chown -R www-data app/tmp
```

Se por alguma razão o CakePHP não conseguir escrever nesta pasta, você será avisado por uma mensagem enquanto estiver em modo de desenvolvimento.

Uma Palavra Sobre o mod_rewrite

Ocasionalmente, um novo usuário irá esbarrar em problemas com o mod_rewrite, então vou abordá-los superficialmente aqui. Se a página de boas-vindas do CakePHP parecer um pouco sem graça (sem imagens, sem cores e sem os estilos css), isso é um indício de que o mod_rewrite provavelmente não esteja funcionando em seu sistema. Aqui estão algumas dicas para lhe ajudar a deixar tudo funcionando corretamente:

1. Certifique-se de que a sobrescrita de opções do .htaccess está habilitada: em seu arquivo httpd.conf, você deve ter uma parte que define uma seção para cada <Directory> do seu servidor. Certifique-se de que a opção `AllowOverride` esteja com o valor `All` para o <Directory> correto. Por questões de segurança e performance, *não* defina `AllowOverride` para `All` em `<Directory />`. Ao invés disso, procure o bloco `<Directory>` que se refere ao seu diretório raíz de seu website.
2. Certifique-se de estar editando o arquivo httpd.conf ao invés de algum específico, que seja válido apenas para um dado usuário ou para um dado site.
3. Por alguma razão, você pode ter obtido uma cópia do CakePHP sem os arquivos .htaccess. Isto algumas vezes acontece porque alguns sistemas operacionais tratam arquivos que começam com "." como arquivos ocultos e normalmente não fazem cópias deles. Certifique-se de obter sua cópia do CakePHP diretamente da seção de downloads do site ou de nosso repositório git.
4. Certifique-se de que o Apache esteja carregando o mod_rewrite corretamente! Você deve ver algo como:

   ```
   LoadModule rewrite_module       libexec/httpd/mod_rewrite.so
   ```

   ou (para o Apache 1.3):

   ```
   AddModule                       mod_rewrite.c
   ```

   em seu httpd.conf.

Se você não quiser ou não puder carregar o mod_rewrite (ou algum outro módulo compatível) em seu servidor, você vai precisar usar o recurso de URLs amigáveis do CakePHP. No arquivo `/app/Config/core.php`, descomente uma linha parecida com:

```
Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

E remova também os arquivos .htaccess em:

```
/.htaccess
/app/.htaccess
/app/webroot/.htaccess
```

Com isto, suas URLs ficarão parecidas com www.exemplo.com/index.php/nomecontroller/nomeaction/param ao invés de www.exemplo.com/nomecontroller/nomeaction/param.

Se você está instalando o CakePHP em outro webserver diferente do Apache, você pode encontrar instruções para ter a reescrita de URLs funcionando na seção *Instalação Avançada*.

Continue lendo este tutorial em *Blog - Continuação* para começar a construir sua primeira aplicação CakePHP.

Blog - Continuação

Crie um Model Post

A classe Model é o pão com manteiga das aplicações CakePHP. Ao criar um model do CakePHP que irá interagir com nossa base de dados, teremos os alicerces necessários para posteriormente fazer nossas operações de visualizar, adicionar, editar e excluir.

Os arquivos de classe do tipo model do CakePHP ficam em `/app/Model` e o arquivo que iremos criar será salvo em `/app/Model/Post.php`. O conteúdo completo deste arquivo deve ficar assim:

```
class Post extends AppModel {
    public $name = 'Post';
}
```

A nomenclatura da classe segue uma convenção que é muito importante no CakePHP. Ao chamar nosso model de Post, o CakePHP pode automaticamente deduzir que este model será usado num PostsController, e que manipulará os dados de uma tabela do banco chamada de `posts`.

Nota: O CakePHP irá criar um objeto (instância) do model dinamicamente para você, se não encontrar um arquivo correspondente na pasta /app/Model. Isto também significa que, se você acidentalmente der um nome errado ao seu arquivo (p.ex. post.php ou posts.php) o CakePHP não será capaz de reconhecer nenhuma de suas configurações adicionais e ao invés disso, passará a usar seus padrões definidos internamente na classe Model.

Para saber mais sobre models, como prefixos de nomes de tabelas, callbacks e validações, confira o capítulo sobre /models deste manual.

Crie o Controller Posts

A seguir, vamos criar um controller para nossos posts. O controller é onde toda a lógica de negócio para interações vai acontecer. De uma forma geral, é o local onde você vai manipular os models e lidar com o resultado das ações feitas sobre nossos posts. Vamos pôr este novo controller num arquivo chamado `PostsController.php` dentro do diretório `/app/Controller`. Aqui está como um controller básico deve se parecer:

```
class PostsController extends AppController {
    public $helpers = array ('Html','Form');
    public $name = 'Posts';
}
```

Agora, vamos adicionar uma action ao nosso controller. Actions quase sempre representam uma única função ou interface numa aplicação. Por exemplo, quando os usuários acessarem o endereço www.exemplo.com/posts/index (que, neste caso é o mesmo que www.exemplo.com/posts/), eles esperam ver a listagem dos posts. O código para tal ação deve se parecer com algo assim:

```
class PostsController extends AppController {
    public $helpers = array ('Html','Form');
    public $name = 'Posts';

    function index() {
        $this->set('posts', $this->Post->find('all'));
    }
}
```

Deixe-me explicar a ação um pouco. Definindo a função `index()` em nosso PostsController, os usuários podem acessar esta lógica visitando o endereço www.exemplo.com/posts/index. De maneira semelhante, se definirmos um método chamado `foobar()` dentro do controller, os usuários deveriam ser capazes de acessá-lo pelo endereço www.exemplo.com/posts/foobar.

Aviso: Você pode ficar tentado a nomear seus controller e actions de uma certa maneira visando obter uma certa URL. Mas resista a esta tentação. Siga as convenções do CakePHP (nomes de controllers no plural, etc) e crie nomes de actions e controllers que sejam legíveis e também compreensíveis. Você sempre vai poder mapear URLs para seu código utilizando «rotas», conforme mostraremos mais à frente.

A única declaração na nossa action utiliza o método `set()` para passar dados do controller para a view (que vamos criar logo mais). A linha define uma variável na view chamada "posts" que vai conter o retorno da chamada do método `find('all')` do model Post. Nosso model Post está automaticamente disponível como `$this->Post` uma vez que seguimos as convenções de nomenclatura do Cake.

Para aprender mais sobre controllers do CakePHP, confira a seção *Controllers*.

Criando as Views de Posts

Agora que temos nossos dados chegando ao nosso model e com a lógica da nossa aplicação definida em nosso controller, vamos criar uma view para a ação `index()` que criamos acima.

As views do Cake são meros fragmentos voltados à apresentação de dados que vão dentro do layout da aplicação. Para a maioria das aplicações, as views serão marcações HTML intercalados com código PHP, mas as views também podem ser renderizadas como XML, CVS ou mesmo como dados binários.

Os layouts são páginas que encapsulam as views e que podem ser intercambiáveis, mas por agora, vamos apenas usar o layout padrão.

Lembra da última seção, em que associamos a variável "posts" para a view usando o método `set()`? Com aquilo, os dados foram repassados para a view num formato parecido com este:

```
// print_r($posts) output:

Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
```

```
                    [title] => The title
                    [body] => This is the post body.
                    [created] => 2008-02-13 18:34:55
                    [modified] =>
                )
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => A title once again
                    [body] => And the post body follows.
                    [created] => 2008-02-13 18:34:56
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Title strikes back
                    [body] => This is really exciting! Not.
                    [created] => 2008-02-13 18:34:57
                    [modified] =>
                )
        )
)
```

Os arquivos de view do Cake são armazenados na pasta `/app/View` dentro de uma pasta com o mesmo nome do controller a que correspondem (em nosso caso, vamos criar uma pasta chamada "Posts"). Para apresentar os dados do post num formato adequado de tabela, o código de nossa view deve ser algo como:

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Posts do Blog</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Título</th>
        <th>Data de Criação</th>
    </tr>

    <!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo as informações␣
↪dos posts -->

    <?php foreach ($posts as $post): ?>
        <tr>
            <td><?php echo $post['Post']['id']; ?></td>
            <td>
                <?php echo $this->Html->link($post['Post']['title'],
                    array('controller' => 'posts', 'action' => 'view', $post['Post'][
↪'id'])); ?>
            </td>
            <td><?php echo $post['Post']['created']; ?></td>
        </tr>
    <?php endforeach; ?>
```

```
</table>
```

Isto é tão simples quanto parece!

Você deve ter notado o uso de um objeto chamado `$this->Html`. Esta é uma instância da classe `HtmlHelper` do CakePHP. O CakePHP vem com um conjunto de helpers que tornam uma moleza fazer coisas como criar links, gerar formulários, Javascript e elementos dinâmicos com Ajax. Você pode aprender mais sobre como usá-los na seção *Helpers*, mas o importante a ser notado aqui é que o método `link()` irá gerar um link em HTML com o título (o primeiro parâmetro) e URL (o segundo parâmetro) dados.

Ao especificar URLs no Cake, é recomendado que você use o formato de array. Este assunto é explicado com mais detalhes na seção sobre Rotas. Usar o formato de array para URLs, permite que você tire vantagens da capacidade do CakePHP de reverter este formato de URL em URLs relativas e vice versa. você também pode simplesmente informar um caminho relativo à base da aplicação na forma /controller/action/parametro_1/parametro_2.

Neste ponto, você deve ser capaz de apontar seu navegador para http://www.exemplo.com/posts/index. Você deve ver sua view, corretamente formatada com o título e a tabela listando os posts.

Se lhe ocorreu clicar num dos links que criamos nesta view (no título do post e que apontam para uma URL /posts/view/algum_id), você provavelmente recebeu uma mensagem do CakePHP dizendo que a action ainda não foi definida. Se você não tiver visto um aviso assim, então ou alguma coisa deu errado ou então você já tinha definido uma action anteriormente, e neste caso, você é muito afoito. Se não, vamos criá-la em nosso PostsController agora:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
    public $name = 'Posts';

    public function index() {
         $this->set('posts', $this->Post->find('all'));
    }

    public function view($id = null) {
        $this->set('post', $this->Post->findById($id));
    }
}
```

A chamada do método `set()` deve lhe parece familiar. Perceba que estamos usando o método `read()` ao invés do `find('all')` porque nós realmente só queremos informações de um único post.

Note que a action de nossa view recebe um parâmetro: O ID do post que queremos ver. Este parâmetro é repassado à action por meio da URL requisitada. Se um usuário acessar uma URL /posts/view/3, então o valor "3" será atribuído ao parâmetro `$id`.

Agora vamos criar a view para nossa nova action "view" e colocá-la em `/app/View/Posts/view.ctp`:

```
<!-- File: /app/View/Posts/view.ctp -->

<h1><?php echo $post['Post']['title']?></h1>

<p><small>Created: <?php echo $post['Post']['created']?></small></p>

<p><?php echo $post['Post']['body']?></p>
```

Confira se está funcionando tentando acessar os links em `/posts/index` ou requisitando diretamente um post acessando `/posts/view/1`.

Adicionando Posts

Ler a partir da base de dados e exibir os posts foi um grande começo, mas precisamos permitir também que os usuários adicionem novos posts.

Primeiramente, comece criando uma action `add()` no PostsController:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form', 'Flash');
    public $components = array('Flash');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }

    public function view($id) {
        $this->set('post', $this->Post->findById($id));
    }

    public function add() {
        if ($this->request->is('post')) {
            if ($this->Post->save($this->request->data)) {
                $this->Flash->success('Your post has been saved.');
                $this->redirect(array('action' => 'index'));
            }
        }
    }
}
```

Nota: Você precisa incluir o componente FlashComponent e o helper FlashHelper em qualquer controller que você manipula variáveis de sessão. Neste caso, incluímos apenas o componente porque ele carrega o helper automaticamente. Se você sempre utiliza sessões, inclua o componente no seu arquivo AppController.

Aqui está o que a action `add()` faz: se o método da requisição feita pelo cliente for do tipo post, ou seja, se ele enviou dados pelo formulário, tenta salvar os dados usando o model Post. Se, por alguma razão ele não salvar, apenas renderize a view. Isto nos dá uma oportunidade de mostrar erros de validação e outros avisos ao usuário.

Quando um usuário utiliza um formulário para submeter (POSTar) dados para sua aplicação, esta informação fica disponível em `$this->request->data`.Você pode usar as funções `pr()` ou `debug()` para exibir os dados se você quiser conferir como eles se parecem.

Nós usamos o método `FlashComponent::success()` do componente FlashComponent para definir uma variável de sessão com uma mensagem a ser exibida na página depois de ser redirecionada. No layout, nós temos `FlashHelper::render()` que exibe a mensagem e limpa a variável de sessão correspondente. O método `Controller::redirect` do controller redireciona para outra URL. O parâmetro `array('action' => 'index')` é convertido para a URL /posts, em outras palavras, a action index do controller posts. Você pode conferir a função `Router::url()` na API para ver os formatos que você pode usar ao especificar uma URL para actions do CakePHP.

Chamar o método `save()` irá verificar por erros de validação e abortar o salvamento se algum erro ocorrer. Vamos falar mais sobre erros de validação e sobre como manipulá-los nas seções seguintes.

Validação de Dados

O CakePHP percorreu uma longa estrada combatendo a monotonia da validação de dados de formulários. Todo mundo detesta codificar formulários intermináveis e suas rotinas de validação. O CakePHP torna tudo isso mais fácil e mais

rápido.

Para usufruir das vantagens dos recursos de validação, você vai precisar usar o FormHelper do Cake em suas views. O `FormHelper` está disponível por padrão em todas as suas views na variável `$this->Form`.

Aqui está nossa view add:

```
<!-- File: /app/View/Posts/add.ctp -->

<h1>Add Post</h1>
<?php
echo $this->Form->create('Post');
echo $this->Form->input('title');
echo $this->Form->input('body', array('rows' => '3'));
echo $this->Form->end('Save Post');
```

Aqui, usamos o FormHelper para gerar a tag de abertura para um formulário. Aqui está o HTML gerado pelo `$this->Form->create()`:

```
<form id="PostAddForm" method="post" action="/posts/add">
```

Se o método `create()` for chamado sem quaisquer parâmetros, o CakePHP assume que você está criando um formulário que submete para a action `add()` do controller atual (ou para a action `edit()` se um campo id for incluído nos dados do formulário), via POST.

O método `$this->Form->input()` é usado para criar elementos de formulário de mesmo nome. O primeiro parâmetro informa ao CakePHP qual o campo correspondente e o segundo parâmetro permite que você especifique um extenso array de opções. Neste caso, o número de linhas para o textarea. Há alguma introspecção «automágica» envolvida aqui: o `input()` irá exibir diferentes elementos de formulário com base no campo do model em questão.

A chamada à `$this->Form->end()` gera um botão de submissão e encerra o formulário. Se uma string for informada como primeiro parâmetro para o `end()`, o FormHelper exibe um botão de submit apropriadamente rotulado junto com a tag de fechamento do formulário. Novamente, confira o capítulo sobre os *Helpers* disponíveis no CakePHP para mais informações sobre os helpers.

Agora vamos voltar e atualizar nossa view `/app/View/Post/index.ctp` para incluir um novo link para «Adicionar Post». Antes de <table>, adicione a seguinte linha:

```
echo $this->Html->link('Add Post', array('controller' => 'posts', 'action' => 'add'));
```

Você pode estar imaginando: como eu informo ao CakePHP sobre os requisitos de validação de meus dados? Regras de validação são definidas no model. Vamos olhar de volta nosso model Post e fazer alguns pequenos ajustes:

```
class Post extends AppModel {
    public $name = 'Post';

    public $validate = array(
        'title' => array(
            'rule' => 'notEmpty'
        ),
        'body' => array(
            'rule' => 'notEmpty'
        )
    );
}
```

O array `$validate` diz ao CakePHP sobre como validar seus dados quando o método `save()` for chamado. Aqui, eu especifiquei que tanto os campos body e title não podem ser vazios. O mecanismo de validação do CakePHP é robusto, com diversas regras predefinidas (números de cartão de crédito, endereços de e-mail, etc.) além de ser

bastante flexível, permitindo adicionar suas próprias regras de validação. Para mais informações, confira o capítulo sobre /models/data-validation.

Agora que você incluiu as devidas regras de validação, tente adicionar um post com um título ou com o corpo vazio para ver como funciona. Uma vez que usamos o método `FormHelper::input()` do FormHelper para criar nossos elementos de formulário, nossas mensagens de erros de validação serão mostradas automaticamente.

Editando Posts

Edição de Posts: Aqui vamos nós. A partir de agora você já é um profissional do CakePHP, então você deve ter identificado um padrão. Criar a action e então criar a view. Aqui está como o código da action `edit()` do PostsController deve se parecer:

```
function edit($id = null) {
    $this->Post->id = $id;
    if ($this->request->is('get')) {
        $this->request->data = $this->Post->findById($id);
    } else {
        if ($this->Post->save($this->request->data)) {
            $this->Flash->success('Your post has been updated.');
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

Esta action primeiro verifica se a requisição é do tipo GET. Se for, nós buscamos o Post e passamos para a view. Se a requisição não for do tipo GET, provavelmente esta contém dados de um formulário POST. Nós usaremos estes dados para atualizar o registro do nosso Post ou exibir novamente a view mostrando para o usuário os erros de validação.

A view edit pode ser algo parecido com isto:

```
<!-- File: /app/View/Posts/edit.ctp -->

<h1>Edit Post</h1>
<?php
    echo $this->Form->create('Post', array('action' => 'edit'));
    echo $this->Form->input('title');
    echo $this->Form->input('body', array('rows' => '3'));
    echo $this->Form->input('id', array('type' => 'hidden'));
    echo $this->Form->end('Save Post');
```

Esta view exibe o formulário de edição (com os valores populados), juntamente com quaisquer mensagens de erro de validação.

Uma coisa a atentar aqui: o CakePHP vai assumir que você está editando um model se o campo "id" estiver presente no array de dados. Se nenhum "id" estiver presente (como a view add de inserção), o Cake irá assumir que você está inserindo um novo model quando o método `save()` for chamado.

Você agora pode atualizar sua view index com os links para editar os posts específicos:

```
<!-- File: /app/View/Posts/index.ctp  (links para edição adicionados) -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link("Add Post", array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
```

```
        <th>Action</th>
        <th>Created</th>
    </tr>

<!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo
as informações dos posts -->

<?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' =>
↪'view', $post['Post']['id']));?>
        </td>
        <td>
            <?php echo $this->Form->postLink(
                'Delete',
                array('action' => 'delete', $post['Post']['id']),
                array('confirm' => 'Are you sure?')
            )?>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post
↪']['id']));?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
<?php endforeach; ?>

</table>
```

Deletando Posts

A seguir, vamos criar uma maneira para os usuários excluírem posts. Comece com uma action `delete()` no PostsController:

```
function delete($id) {
    if (!$this->request->is('post')) {
        throw new MethodNotAllowedException();
    }
    if ($this->Post->delete($id)) {
        $this->Flash->success('The post with id: ' . $id . ' has been deleted.');
        $this->redirect(array('action' => 'index'));
    }
}
```

Esta lógica exclui o post dado por $id, e utiliza `$this->Flash->success()` para mostrar uma mensagem de confirmação para o usuário depois de redirecioná-lo para `/posts`.

Se o usuário tentar deletar um post usando uma requisição do tipo GET, nós lançamos uma exceção. Exceções não apanhadas são capturadas pelo manipulador de exceções do CakePHP e uma página de erro amigável é mostrada. O CakePHP vem com muitas *Exceptions* que você pode usar para indicar vários tipos de erros HTTP que sua aplicação pode precisar gerar.

Como estamos executando apenas uma lógica de negócio e redirecionando, esta action não tem uma view. Você pode querer atualizar sua view index com links que permitam ao usuários excluir posts, porém, como um link executa uma requisição do tipo GET, nossa action irá lançar uma exceção. Precisamos então criar um pequeno formulário que enviará um método POST adequado. Para estes casos o helper FormHelper fornece o método `postLink()`:

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link('Add Post', array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Actions</th>
        <th>Created</th>
    </tr>

<!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo as informações dos␣
↪posts -->

    <?php foreach ($posts as $post): ?>
        <tr>
            <td><?php echo $post['Post']['id']; ?></td>
            <td>
                <?php echo $this->Html->link($post['Post']['title'], array('action' =>
↪ 'view', $post['Post']['id']));?>
            </td>
            <td>
                <?php echo $this->Form->postLink(
                    'Delete',
                    array('action' => 'delete', $post['Post']['id']),
                    array('confirm' => 'Are you sure?'));
                ?>
            </td>
            <td><?php echo $post['Post']['created']; ?></td>
        </tr>
    <?php endforeach; ?>

</table>
```

Nota: O código desta view também utiliza o HtmlHelper para solicitar uma confirmação do usuário com um diálogo em Javascript antes de tentar excluir o post.

Rotas

Para alguns, o roteamento padrão do CakePHP funcionará muito bem. Os desenvolvedores que estiverem mais afeitos a criar produtos ainda mais amigáveis aos usuários e aos mecanismos de busca irão gostar da maneira que as URLs do CakePHP são mapeadas para actions específicas. Então vamos fazer uma pequena alteração de rotas neste tutorial.

Para mais informações sobre técnicas avançadas de roteamento, veja *Routes Configuration*.

Por padrão, o CakePHP responde a requisições para a raiz de seu site (i.e. http://www.exemplo.com) usando seu PagesController e renderizando uma view chamada de «home». Ao invés disso, vamos substituir isto por nosso PostsController criando uma regra de roteamento.

As rotas do Cake são encontrada no arquivo `/app/Config/routes.php`. Você vai querer comentar ou remover a linha que define a rota raiz padrão. Ela se parece com:

```
Router::connect('/', array('controller' => 'pages', 'action' => 'display', 'home'));
```

Esta linha conecta a URL "/" com a home page padrão do CakePHP. Queremos conectá-la com nosso próprio controller, então adicionamos uma linha parecida com isto:

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

Isto deve conectar as requisições de "/" à action `index()` que criaremos em nosso PostsController.

Nota: O CakePHP também faz uso do "roteamento reverso" - se, com a rota definida acima, você passar `array('controller' => 'posts', 'action' => 'index')` a um método que espere um array, a URL resultante será "/". É sempre uma boa ideia usar arrays para URLs, já que é a partir disto que suas rotas definem para onde suas URLs apontam, além de garantir que os links sempre apontem para o mesmo lugar.

Conclusão

Criar aplicações desta maneira irá lhe trazer paz, honra, amor e dinheiro além de satisfazer às suas mais ousadas fantasias. Simples, não? Tenha em mente que este tutorial foi muito básico. O CakePHP possui *muito* mais recursos a oferecer e é flexível de tantas maneiras que não conseguimos mostrar aqui por questões de simplicidade. Utilize o resto deste manual como guia para construir mais aplicações ricas em recursos.

Agora que você criou uma aplicação básica com o Cake, você está pronto para a coisa real. Comece seu próprio projeto, leia o restante do Manual e da API[2].

E se você precisar de ajuda, nos vemos no canal #cakephp (e no #cakephp-pt). Seja bem-vindo ao CakePHP!

Leitura Recomendada

Estas são as tarefas comuns que pessoas aprendendo o CakePHP geralmente querem estudar:

1. *Layouts*: Customizando o layout do seu website
2. *Elements* Incluindo e reutilizando trechos de código
3. *Scaffolding (arcabouços)*: Prototipando antes de programar
4. /console-and-shells/code-generation-with-bake Gerando código CRUD básico
5. *Autenticação simples e Autorização da Aplicação*: Tutorial de autenticação e autorização de usuários

Leitura Adicional

Uma Requisição Típica do CakePHP

Nós já abordamos os ingredientes básicos do CakePHP, então agora vamos dar uma olhada em como os objetos trabalham juntos para completar uma requisição básica. Continuando com o exemplo da requisição original, vamos imaginar que nosso amigo Ricardo apenas clicou no link «Compre um bolo personalizado agora!» em uma landing page[3] de uma aplicação CakePHP.

Figure: 2. Requisição típica do CakePHP.

Em preto = elemento requerido, em cinza = elemento opcional, em Azul = callbacks

1. Ricardo clica no link apontando para http://www.example.com/cakes/buy, e o navegador dele faz uma requisição para seu servidor web.

[2] https://api.cakephp.org/2.x/
[3] http://pt.wikipedia.org/wiki/Landing_page

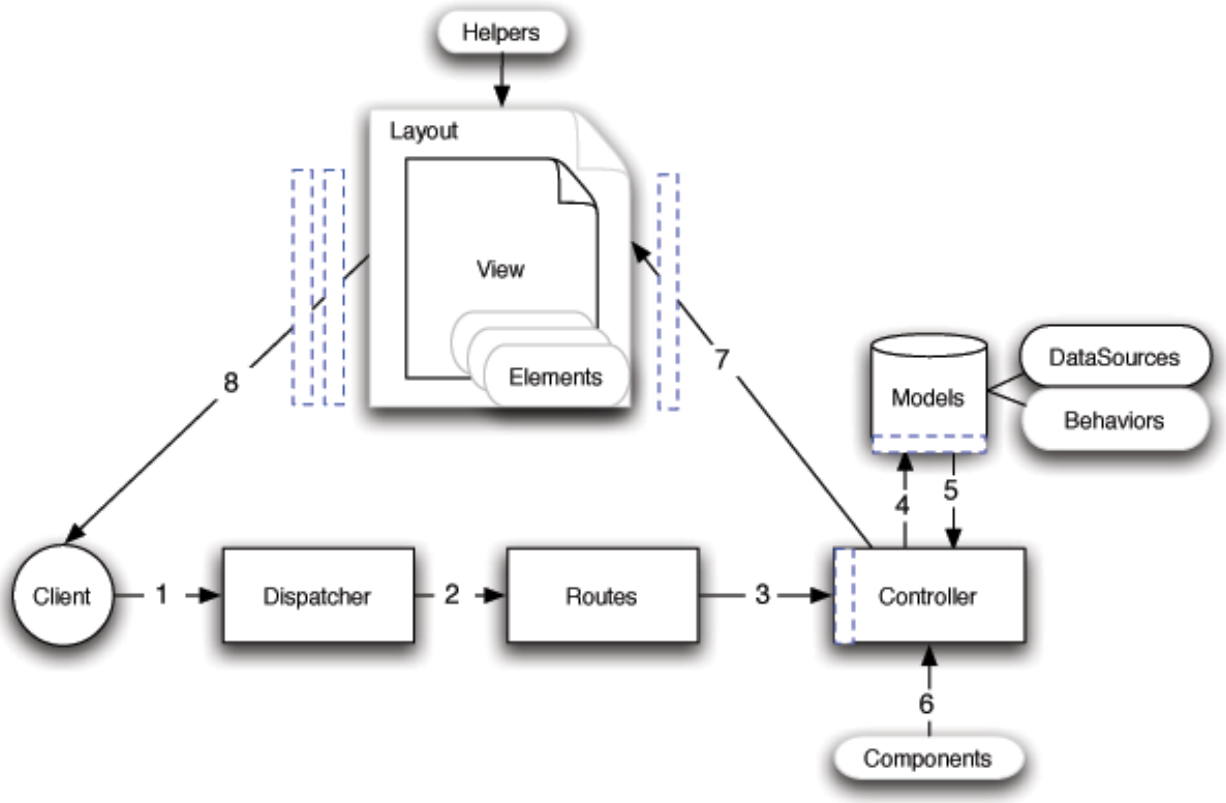

Fig. 1.1: Um diagrama de fluxo mostrando uma típica requisição CakePHP

2. O roteador analisa a URL para extrair os parâmetros desta requisição: o controller, a ação, e qualquer outro argumento que afeta a lógica do negócio durante esta requisição.
3. Usando rotas, a URL da requisição é mapeada para uma ação de um controller (um método em uma classe controller específica). Neste caso, será o método buy() do controller CakesController. O callback beforeFilter() do controller é chamado antes de qualquer ação lógica do controller. As linhas tracejadas em azul mostram isso no diagrama.
4. O controller pode usar models para obter acesso aos dados do aplicativo. Neste exemplo o controller usa o model para pegar no banco de dados as últimas compras do Ricardo. Qualquer callback do model, behaviors ou DataSources que for aplicável neste momento, será chamado. Enquanto a utilização de Models não seja obrigatória, todas os controllers inicialmente requerem ao menos um model.
5. Após o model buscar os dados, estes são retornados para o controller. Callbacks de um Model podem ser aplicados.
6. O controller poderá utilizar componentes para refinar os dados ou executar outras operações (manipular sessão, autenticar ou enviar e-mails são exemplos)
7. Uma vez que o controller usou os models e componentes para preparar os dados de forma suficiente, os dados são passados para a view usando o método set() do controller. Callbacks do controller podem ser chamados antes que os dados sejam passados. A view é executada, podendo incluir o uso de elementos e/ou helpers. Por padrão, a view é renderizada dentro de um layout.
8. Adicionalmente, callbacks do controller (como o afterFilter) podem ser aplicados. A view renderizada e completa é enviada para o navegador do Ricardo.

Convenções no CakePHP

Nós somos grandes fãns de convenções sobre configuração[4]. Enquanto pode levar um tempo para aprender as convenções do CakePHP, você ganhará muito tempo a longo prazo: seguindo as convenções você ganhará funcionalidades e ficará livre dos pesadelos de manter arquivos de configuração. As convenções também contribuem para o desenvolvimento de sistemas mais uniformes, permitindo que outros desenvolvedores entrem no projeto e comecem a trabalhar muito mais rapidamente.

As convenções do CakePHP foram destiladas ao longo de anos de experiência no desenvolvimento de aplicações web e boas práticas. Da mesma forma que sugerimos que você use essas convenções enquanto desenvolve com o CakePHP, devemos mencionar que muitos destes princípios são facilmente sobrescritos – algo que é especialmente útil quando se trabalha com sistemas legados.

Convenções nos Controllers

As classes Controllers devem ser escritas no plural, usando o formato CamelCase[5] e terminarem com a palavra `Controller`. `PeopleController` e `LatestArticlesController` são dois exemplos de nomes de controllers que seguem a convenção.

O primeiro método que você pode escrever para um controller é o método `index()`. Quando uma requisição especifica o controller mas não a ação, o comportamento padrão do CakePHP é executar o método `index()`. Por exemplo, uma requisição para http://www.example.com/apples/ é mapeada para o método `index()` do controller `ApplesController`, enquanto a URL http://www.example.com/apples/view/ é mapeada para a chamada do método `view()` do mesmo controller.

Você também pode alterar a visibilidade dos métodos de controllers no CakePHP prefixando os nome dos métodos com underscores. Se um método de um controller for prefixado, o método não poderá ser acessado diretamente da web mas estará disponível para uso interno. Por exemplo:

[4] http://pt.wikipedia.org/wiki/Conven%C3%A7%C3%A3o_sobre_configura%C3%A7%C3%A3o
[5] http://pt.wikipedia.org/wiki/CamelCase

```
class NewsController extends AppController {

    function latest() {
        $this->_findNewArticles();
    }

    function _findNewArticles() {
        // lógica para encontrar os os últimos artigos
    }
}
```

Enquanto a página http://www.example.com/news/latest/ pode ser acessada normalmente pelos usuários, alguem tentando visitar a página http://www.example.com/news/_findNewArticles/ receberá um erro porque o nome do método é prefixado com um underscore. Você também pode utilizar as palavras-chave de visibilidade do PHP para indicar se um método pode ou não ser acessado por uma URL. Métodos privados não podem ser acessados.

Considerações sobre URLs para nomes de Controllers

Como você viu, controllers com nomes formados por uma palavra são mapeados por URLs em caixa baixa. Por exemplo, `ApplesController` (que seria definido em um arquivo com o nome de "ApplesController.php") pode ser acessado com a seguinte URL: http://example.com/apples.

Controllers formados por mais de uma palavra *podem* ter qualquer forma "flexionada" do nome:

- /redApples
- /RedApples
- /Red_apples
- /red_apples

Todas serão resolvidas para o método index do controller RedApples. Porém, a convenção diz que suas URLs devem ser em caixa baixa e usar underscores, portanto /red_apples/go_pick é a forma mais apropriada para acessar a ação `RedApplesController::go_pick`.

Para mais informações sobre as URLs do CakePHP e o tratamento de parâmetros, veja *Routes Configuration*.

Convenções de Arquivos e Nomes de Classes

Geralmente, nomes de arquivos correspondem com o nome de suas classes, que são em CamelCase[6]. Então se você possui uma classe **MyNiftyClass**, para o Cake, o nome do arquivo deve ser **MyNiftyClass.php**. Abaixo estão alguns exemplos de como nomear arquivos para diferentes tipos de classes que você usará em aplicações CakePHP:

- O controller **KissesAndHugsController** seria encontrado em um arquivo chamado **KissesAndHugsController.php**
- O componente **MyHandyComponent** seria encontrado em um arquivo chamado **MyHandyComponent.php**
- O model **OptionValue** seria encontrado em um arquivo chamado **OptionValue.php**
- O behavior **EspeciallyFunkableBehavior** seria encontrado em um arquivo chamado **EspeciallyFunkableBehavior.php**
- A View **SuperSimpleView** seria encontrado em um arquivo chamado **SuperSimpleView.php**
- O helper **BestEverHelper** seria encontrado em um arquivo chamado **BestEverHelper.php**

[6] http://pt.wikipedia.org/wiki/CamelCase

Cada arquivo deverá estar em uma pasta apropriada no diretório app da sua aplicação.

Convenções de Models e Banco de Dados

O nome dos Models devem ser escritos no singular e no formato CamelCase[7]. Car, BigCar e ReallyBigCar são todos exemplos de nomes de models que seguem a convenção.

Nomes de tabelas correspondentes à models do CakePHP são escritos no plural e usando underscores. As tabelas correspondentes para os models mencionados acima são respectivamente cars, `big_cars` e `really_big_cars`.

Você pode usar a biblioteca utilitária `Inflector` para verificar a forma singular/plural das palavras. Veja a classe /core-utility-libraries/inflector para mais informações.

Nomes de colunas formadas por mais de uma palavra devem ser separadas usando underscore como em first_name.

Chaves estrangeiras em associações do tipo hasMany, belongsTo ou hasOne são reconhecidas por padrão como o nome (no singular) das tabelas relacionadas seguidas por _id. Então, se Baker hasMany (possui muitos) Cake, a tabela cakes irá fazer referência a tabela bakers via chave estrangeira baker_id. Para tabelas formadas por mais de uma palavra como category_types, a chave estrangeira seria category_type_id.

Tabelas de junções usadas em relacionamentos do tipo hasAndBelongsToMany (HABTM) entre models devem ser nomeadas usando o nome das tabelas dos models referenciados unidas em ordem alfabética (apples_zebras ao invés de zebras_apples).

Todas as tabela com que models do CakePHP interagem (com exceção das tabelas de junção) requerem uma chave primária para identificar unicamente cada registro. Se você quiser modelar uma tabela que não possua uma chave primária única, a convenção do CakePHP diz que você deve adicionar uma se quiser utilizá-la com um model.

O CakePHP não suporta chaves primárias compostas. Se você quiser manipular os dados das tabelas de junções diretamente, use chamadas de query diretas ou adicione uma chave primaria para usá-las como um model normal. Ex.:

```
CREATE TABLE posts_tags (
    id      INT(10) NOT NULL AUTO_INCREMENT,
    post_id INT(10) NOT NULL,
    tag_id  INT(10) NOT NULL,
    PRIMARY KEY(id)
);
```

Ao invés de usar chaves auto incrementadas, você também pode usar o tipo char(36). Desta forma o Cake irá usar um identificador único (uuid) de 36 caracteres criado por String::uuid sempre que você salvar um novo registro usando o método Model::save.

Convenções de Views

Arquivos de templates de views são nomeados de acordo com o nome do método do controller que exibem no formato underscore. O método getReady() da classe PeopleController irá utilizar uma view localizada em /app/View/People/get_ready.ctp.

O molde padrão é /app/View/Controller/underscored_function_name.ctp.

Nomeando as partes de sua aplicação usando as convenções do CakePHP, você ganha funcionalidades sem os incômodos e problemáticos arquivos de configuração. Segue agora um exemplo final que mostra as convenções todas juntas.

- Tabela do banco de dados: «cars»
- Classe Model: «Car», encontrada em /app/Model/Car.php

[7] http://pt.wikipedia.org/wiki/CamelCase

- Classe Controller: «CarsController», encontrada em /app/Controller/CarsController.php
- Arquivo de View encontrada em /app/View/Cars/index.ctp

Usando estas convenções o CakePHP saberá que uma requisição feita pela URL http://example.com/cars/ refere-se a uma chamada para o método index() da classe CarsController, onde o model Car é automaticamente disponibilizado (e automaticamente amarrado com a tabela cars no banco de dados) e renderiza o arquivo /app/View/Cars/index.ctp. Nenhum destes relacionamentos precisou ser configurado, a não ser a criação de classes e arquivos que você precisaria criar de qualquer maneira.

Agora que você já foi introduzido aos fundamentos do CakePHP, você pode tentar o *Blog* para ver como todas as coisas se encaixam juntas.

Estrutura de Diretórios no CakePHP

Após ter baixado e extraído o CakePHP, você deverá encontrar os seguintes arquivos e pastas:

- app
- lib
- vendors
- plugins
- .htaccess
- index.php
- README

Você encontrará três pastas principais:

- No diretório *app* é onde você fará sua mágica, ou seja: é o lugar que você colocará os arquivos de sua aplicação.
- No diretório *lib* é onde fazemos nossa mágica. comprometa-se em não editar nenhum arquivo deste diretório. Não podemos ajudá-lo se você modificar o núcleo do framework.
- E finalmente, no diretório *vendors* é onde você pode colocar as bibliotecas de terceiros que precisar usar em suas aplicações com o CakePHP.

O Diretório App

No diretório app do Cake é onde você faz a maior parte do desenvolvimento de sua aplicação. Vamos dar uma olhada mais de perto nas pastas que estão dentro de app.

Config Armazena os (poucos) arquivos de configuração que o CakePHP utiliza. Parâmetros de conexão com o banco de dados, inicialização do sistema (Bootstrapping[8]), arquivos de configuração do núcleo do framework, e outros, devem ficar aqui.

Controller Contém os controllers e componentes da sua aplicação.

Lib Contém suas bibliotecas pessoais e diferentes das obtidas de terceiros. Isto permite separar as bibliotecas internas de sua empresa das que foram criadas por outras pessoas ou fornecedores.

Locale Armazena arquivos contendo strings de internacionalização.

Model Contém os Models, behaviors e datasources de sua aplicação.

Plugin Contém pacotes de plugins.

[8] http://pt.wikipedia.org/wiki/Bootstrapping

tmp Este diretório é onde o CakePHP armazena dados temporários. Os dados armazenados dependem de como você configurou o CakePHP mas geralmente é usada para armazenar o cache das descrições dos models, logs e por vezes os dados de sessão.

Tenha certeza de que esta pasta exista e que seja gravável, senão o desempenho de sua aplicação será prejudicado severamente.

Vendor Qualquer classe ou biblioteca de terceiros devem ficar aqui. Fazendo isto, torna fácil acessá-las usando o método App::import("vendor", "name"). Olhos aguçados notaram que isto parece redundante, já que temos outra pasta chamada vendors, um nível acima do diretório app. Nós entraremos nos detalhes, explicando a diferença dos dois diretórios quando estivermos discutindo sobre como gerir múltiplas aplicações e configurações de sistemas mais complexos.

View Arquivos de apresentação são colocados aqui: elementos, páginas de erros, helpers, layouts e arquivos de views.

webroot Quando você configurar sua aplicação para rodar em produção, este diretório deve ser a raiz do seu diretório web público. Pastas aqui dentro também servem para colocar seus arquivos CSS, imagens e Javascripts.

Estrutura do CakePHP

O CakePHP possui as classes essenciais Controller, Model e View, mas também apresenta algumas outras classes e objetos adicionais que fazem o desenvolvimento com o MVC um pouco mais rápido e divertido. Componentes, Behaviors e Helpers são classes que fornecem extensibilidade e reusabilidade para adicionar funcionalidades à base das classes do MVC em sua aplicação. Por enquanto, vamos fazer uma explicação superficial destas ferramentas e detalhá-las mais tarde.

Extensões de Aplicação

Cada controller, helper e model possui uma classe mãe que você pode usar para incluir mudanças válidas por toda a aplicação. As classes AppController (localizada em `/app/Controller/AppController.php`), AppHelper (localizada em `/app/View/Helper/AppHelper.php`) e AppModel (localizada em `/app/Model/AppModel.php`) são excelentes lugares para colocar métodos que você quer compartilhar entre todos os controllers, helpers ou models.

Embora rotas não sejam classes ou arquivos, elas desempenham um papel nas requisições feitas ao CakePHP. Definições de rotas dizem ao CakePHP como mapear URLs para as ações de controllers. O comportamento padrão assume que a URL `/controller/action/var1/var2` deve ser mapeada para o método Controller::action($var1, $var2), mas você pode usar as rotas para customizar as URLs e como elas são interpretadas por sua aplicação.

Alguns recursos em uma aplicação merecem ser reunidas em um pacote. Um plugin é um pacote de models, controllers e views que cumprem um objetivo específico e que podem ser utilizados em várias aplicações. Um sistema de gerenciamento de usuários ou um blog simplificado podem ser bons candidatos para plugins do CakePHP.

Extensões de Controllers («Componentes»)

Um *Component* é uma classe que dá suporte às lógicas nos controllers. Se você possui uma lógica que queira compartilhar entre controllers, um componente geralmente é uma boa escolha para colocá-la. Como um exemplo, a classe EmailComponent do Cake permite criar e enviar emails num piscar de olhos.

Ao invés de escrever um método em um controller único que executa esta lógica, você pode empacotar a lógica para que seja possível compartilhá-la.

Os controllers também estão equipados com callbacks. Estes callbacks estão disponíveis para que você possa utilizá-los, bem nos casos em que você precisa inserir alguma lógica entre as operações do núcleo do CakePHP. Os callbacks disponibilizados são:

- `beforeFilter()`, executado antes de qualquer ação de um controller.
- `beforeRender()`, executado após a lógica de um controller, mas antes da view ser renderizada.
- `afterFilter()`, executada após a lógica de um controller, incluindo a renderização da view. Pode não haver diferenças entre o `afterRender()` e o `afterFilter()` ao menos que você tenha chamado o método `render()` na ação de um controller e tenha incluído alguma lógica depois desta chamada.

Extensões de Models («Behaviors»)

Similarmente, Behaviors trabalham para adicionar funcionalidades comuns entre models. Por exemplo, se você armazena os dados dos usuários em uma estrutura de dados do tipo árvore, você pode especificar que seu model *Usuario* se comporta tal como uma árvore e assim, ganha funcionalidades para remover, adicionar e substituir nós na estrutura que existe por baixo do model.

Models também recebem o suporte de outra classe chamada DataSource. DataSources são uma abstração que permite os models manipularem consistentemente diferentes tipos de dados. Embora a fonte principal de dados em uma aplicação usando o CakePHP seja banco de dados, você pode escrever DataSources adicionais que permitem seus models representarem feeds RSS, arquivos CSV, entradas LDAP ou eventos do iCal. Os DataSources permitem você associar registros de diferentes fontes: Diferente de estar limitado pelas junções do SQL, os DataSources permitem você dizer para seu Model LDAP que está associado à muitos eventos do iCal.

Assim como os controllers, os models também possuem callbacks:

- `beforeFind()`, executado antes de uma busca.
- `afterFind()`, executado após uma busca.
- `beforeValidate()`, executado antes de fazer uma validação de dados.
- `beforeSave()`, executado antes de salvar ou atualizar registros de um model.
- `afterSave()`, executado após salvar ou atualizar registros de um model.
- `beforeDelete()`, executado antes de remover registros de um model.
- `afterDelete()`, executado após remover registros de um model.

Com a mínima descrição dada, deve ser possível saber o que estes callbacks fazem. Você pode encontrar mais detalhes no capítulo dos models.

Extensões de Views («Helpers»)

Um Helper é uma classe que ajuda na lógica das views. Muito parecido como os componentes que são usados pelos controllers, os helpers ajudam na lógica de apresentação que podem ser acessadas e compartilhadas entre as views. Um dos Helpers que acompanha o Cake é o AjaxHelper que torna requisições em ajax nas views muito mais fácil.

Muitas aplicações possuem pedaços de código de apresentação que são usados repetidamente. O CakePHP facilita a reutilização destes trechos com layouts e elementos. Por padrão, cada view renderizada por um controller é colocada dentro de um layout. Elementos são usados quando pequenos trechos de conteúdo precisam ser reusados em muitas views.

CAPÍTULO 2

Instalação

O CakePHP é rápido e fácil de instalar. Os requisitos mínimos são um servidor web e uma cópia do Cake, só isso! Apesar deste manual focar principalmente na configuração do Apache (porque ele é o mais comum), você pode configurar o Cake para executar em diversos servidores web, tais como lighttpd ou Microsoft IIS.

Requisitos

- Servidor HTTP. Por exemplo: Apache. É preferível ter o mod_rewrite habilitado mas não é uma exigência.
- PHP 5.2.8 ou superior.

Tecnicamente não é exigido um banco de dados mas imaginamos que a maioria das aplicações irá utilizar um. O CakePHP suporta uma variedade deles:

- MySQL (4 ou superior)
- PostgreSQL
- Microsoft SQL Server
- SQLite

Nota: Todos os drivers inclusos internamente requerem o PDO. Você deve ter certeza que possui a extensão correta do PDO instalada.

Licença

O CakePHP é licenciado sob uma Licença MIT. Isto significa que você tem liberdade para modificar, distribuir e republicar o código-fonte com a condição de que os avisos de *copyright* permaneçam intactos. Você também tem

liberdade para incorporar o CakePHP em qualquer aplicação comercial ou de código fechado.

Baixando o CakePHP

Há duas maneiras de se obter uma cópia atualizada do CakePHP. Você pode fazer o download de um arquivo comprimido (zip/tar.gz/tar.bz2) no site principal ou obter o código a partir do repositório git.

Para fazer o download da versão estável mais recente do CakePHP, visite o site principal https://cakephp.org. Lá haverá um link chamado "Download Now!" para baixar.

Todas as versões liberadas do CakePHP estão hospedadas no Github[9]. O Github do CakePHP abriga o próprio Cake assim como muitos outros plugins para ele. As versões disponíveis estão na página Github tags[10].

Alternativamente você pode obter uma cópia contendo todas as correções de bugs e atualizações recentes clonando o repositório do Github:

```
git clone -b 2.x git://github.com/cakephp/cakephp.git
```

Permissões

O CakePHP usa o diretório `/app/tmp` para diferentes operações. Descrições do modelo, cache das *views*, e informações das sessões são alguns exemplos.

Assim, tenha certeza que o diretório `/app/tmp` na sua instalação do cake permite a escrita pelo usuário do servidor web.

Um problema comum é que ambos os diretórios e subdiretórios de app/tmp devem poder ser gravados pelo servidor web e pelo usuário da linha de comando. Em um sistema UNIX, se o seu usuário do servidor web é diferente do seu usuário da linha de comando, você pode pode executar os seguintes comandos apenas uma vez em seu projeto para assegurar que as permissões serão configuradas apropriadamente:

```
HTTPDUSER=`ps aux | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v␣
↪root | head -1 | cut -d\  -f1`
setfacl -R -m u:${HTTPDUSER}:rwx app/tmp
setfacl -R -d -m u:${HTTPDUSER}:rwx app/tmp
```

Configuração

Configurar o CakePHP pode ser tão simples como descompactá-lo em seu servidor web, ou tão complexo e flexível se você desejar. Esta seção irá cobrir três principais tipos de instalação do CakePHP: desenvolvimento, produção e avançada.

- Desenvolvimento: fácil para começar, as URLs da aplicação incluem o nome do diretório de instalação e é menos seguro.
- Produção: Requer maior habilidade para configurar o diretório raiz do servidor web, URLs limpas, muito seguro.
- Avançada: Com algumas configurações, permite que você coloque os diretórios do CakePHP em diferentes locais do sistema de arquivos, permitindo compartilhar o núcleo do CakePHP entre diversas aplicações.

[9] https://github.com/cakephp/cakephp
[10] https://github.com/cakephp/cakephp/tags

Desenvolvimento

A instalação de desenvolvimento é o método mais rápido de configuração do Cake. Este exemplo irá te ajudar a instalar uma aplicação CakePHP e torná-la disponível em http://www.example.com/cake_2_0/. Assumimos, para efeitos deste exemplo que a sua raiz do documento é definido como /var/www/html.

Descompacte o conteúdo do arquivo do Cake em `/var/www/html`. Você agora tem uma pasta na raiz do seu servidor web com o nome da versão que você baixou (por exemplo, cake2.0.0). Renomeie essa pasta para cake_2_0. Sua configuração de desenvolvimento será semelhante a esta em seu sistema de arquivos:

```
/var/www/html/
    cake_2_0/
        app/
        lib/
        plugins/
        vendors/
        .htaccess
        index.php
        README
```

Se o seu servidor web está configurado corretamente, agora você deve encontrar sua aplicação Cake acessível em http://www.example.com/cake_2_0/.

Utilizando um pacote CakePHP para múltiplas Aplicações

Se você está desenvolvendo uma série de aplicações, muitas vezes faz sentido que elas compartilhem o mesmo pacote. Existem algumas maneiras em que você pode alcançar este objetivo. Muitas vezes, o mais fácil é usar o PHP `include_path`. Para começar, clone o CakePHP em um diretório. Para esse exemplo, nós vamos utilizar `/home/mark/projects`:

```
git clone git://github.com/cakephp/cakephp.git /home/mark/projects/cakephp
```

Isso ira clonar o CakePHP no seu diretório `/home/mark/projects`. Se você não quiser utilizar git, você pode baixar um compilado e os próximos passos serão os mesmos. Em seguida você terá que localizar e modificar seu `php.ini`. Em sistemas *nix está localizado na maioria das vezes em `/etc/php.ini`, mas utilizando `php -i` e procurando por "Loaded Configuration File", você pode achar a localização atual. Uma vez que você achou o arquivo ini correto, modifique a configuração `include_path` para incluir `/home/mark/projects/cakephp/lib`. Um exemplo semelhamte deveria ser como:

```
include_path = .:/home/mark/projects/cakephp/lib:/usr/local/php/lib/php
```

Depois de reiniciar seu servidor web, você deve ver as mudanças refletidas em `phpinfo()`.

Nota: Se você estiver no Windows, separe os caminhos de inclusão com ; ao invés de :

Finalizando a definição do seu `include_path` suas aplicações devem estar prontas para encontrar o CakePHP automaticamente.

Produção

A instalação de produção é uma forma mais flexível de configuração do Cake Usando este método permite um total domínio para agir como uma única aplicação CakePHP. Este exemplo irá ajudá-lo a instalar o Cake em qualquer lugar

do seu sistema de arquivos e torná-lo disponível em http://www.example.com. Note que esta instalação pode requerer os privilégios para alteração do DocumentRoot do servidor Apache.

Descompacte o conteúdo do arquivo do Cake em um diretório de sua escolha. Para fins deste exemplo, assumimos que você escolheu instalar o Cake em /cake_install. Sua configuração de produção será semelhante a esta em seu sistema de arquivos:

```
/cake_install/
    app/
        webroot/ (esse diretório está definido como diretiva ``DocumentRoot``)
    lib/
    plugins/
    vendors/
    .htaccess
    index.php
    README
```

Desenvolvedores usando o Apache devem definir o `DocumentRoot` do domínio para:

```
DocumentRoot /cake_install/app/webroot
```

Se o seu servidor web estiver configurado corretamente, você deve encontrar agora sua aplicação Cake acessível em http://www.example.com.

Instalação Avançada e Configuração Específica por Servidor

Instalação Avançada

Pode haver algumas situações onde você deseja colocar os diretórios do CakePHP em diferentes locais no sistema de arquivos. Isto pode ser devido a uma restrição do servidor compartilhado, ou talvez você queira apenas que algumas aplicações compartilhem as bibliotecas do Cake. Esta seção descreve como espalhar seus diretórios do CakePHP em um sistema de arquivos.

Em primeiro lugar, note que há três partes principais de uma aplicação CakePHP.

1. As bibliotecas do núcleo do CakePHP, em /cake.
2. O código da sua aplicação, em /app.
3. Os arquivos públicos da sua aplicação, normalmente em /app/webroot.

Cada um desses diretórios podem ser localizados em qualquer em seu sistema de arquivos, com exceção do webroot, que precisa ser acessível pelo seu servidor web. Você pode até mesmo mover a pasta webroot para fora da pasta app, desde que você diga ao Cake onde você colocou.

Para configurar sua instalação do Cake, você precisa fazer algumas modificações nos seguintes arquivos.

- /app/webroot/index.php
- /app/webroot/test.php (se você utilizar o recurso de *Testes*.)

Há três constantes que você precisa editar: `ROOT`, `APP_DIR`, e `CAKE_CORE_INCLUDE_PATH`.

- `ROOT` deve ser configurada para o diretório que contém sua pasta app.
- `APP_DIR` deve ser definida como o nome de sua pasta app.
- `CAKE_CORE_INCLUDE_PATH` deve ser definida como o caminho da sua pasta de bibliotecas do CakePHP.

Vamos fazer um exemplo para que você possa ver como funciona uma instalação avançada na prática. Imagine que eu quero que a aplicação funcione como segue:

- O núcleo do CakePHP será colocado em /usr/lib/cake.
- O diretório webroot da minha aplicação será /var/www/mysite/.
- O diretório app da minha aplicação será /home/me/myapp.

Dado este tipo de configuração, eu preciso editar meu arquivo webroot/index.php (que vai acabar em /var/www/mysite/index.php, neste exemplo) para algo como o seguinte:

```
// /app/webroot/index.php (parcial, comentários removidos)

if (!defined('ROOT')) {
    define('ROOT', DS . 'home' . DS . 'me');
}

if (!defined('APP_DIR')) {
    define ('APP_DIR', 'myapp');
}

if (!defined('CAKE_CORE_INCLUDE_PATH')) {
    define('CAKE_CORE_INCLUDE_PATH', DS . 'usr' . DS . 'lib');
}
```

Recomenda-se a utilização da constante `DS` ao invés das barras para delimitar os caminhos de arquivos. Isso previne qualquer erros sobre falta de arquivos que você pode obter, por ter usado o delimitador errado, e isso torna o seu código mais portável.

Apache e mod_rewrite (e .htaccess)

O CakePHP é desenvolvido para trabalhar com o mod_rewrite, mas percebemos que alguns usuários apanharam para fazer isto funcionar nos seus sistemas, então nós lhe daremos algumas dicas que você pode tentar fazer para rodar corretamente.

Aqui estão algumas coisas que você pode tentar fazer para rodar corretamente. Primeiro veja o seu httpd.conf (tenha certeza de estar editando o httpd.conf do sistema e não o de um usuário ou de um site específico).

1. Tenha certeza que a sobreposição do .htaccess está sendo permitida, ou seja, que o AllowOverride está configurado como All para o *DocumentRoot*. Você deve ver algo similar a isso:

```
# Cada diretório com o Apache tenha acesso pode ser configurado com
# relação aos quais serviços e recursos são permitidos e/ou
# desabilitados neste diretório (e seus subdiretórios).
#
# Primeiro, configuramos o o "padrão" para ter um conjunto muito
# restrito de recursos.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Tenha certeza de estar carregando o mod_rewrite corretamente. Você deve ver algo como:

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

Em muitos sistemas isso vem comentado por padrão (começando com um #), então você apenas precisa remover esses símbolos.

Depois de fazer as alterações, reinicie o Apache para ter certeza que as configurações estão aivas.

Verifique se os seus arquivos .htaccess estão nos diretórios corretos.

Isso pode acontecer durante a cópia, pois alguns sistemas operacionais tratam arquivos que começam com "." como oculto e, portanto, você não poderá vê-los copiar.

3. Tenha certeza que sua cópia do CakePHP é veio da seção de downloads do nosso site ou do nosso repositório GIT, e foi descompactada corretamente verificando os seus arquivos .htaccess.

   No diretório raiz do Cake (precisa ser copiado para o seu DocumentRoot, este redireciona tudo para a sua aplicação):

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

   O diretório app do seu Cake (será copiado para o diretório principal da sua aplicação pelo bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule    ^$    webroot/    [L]
    RewriteRule    (.*) webroot/$1    [L]
 </IfModule>
```

   Diretório webroot do Cake (será copiado para a raiz da sua aplicação web pelo bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

   Para muitos serviços de hospedagem (GoDaddy, 1and1), seu servidor web sendo servido a partir de um diretório de usuário que já utiliza o mod_rewrite. Se você está instalando o CakePHP dentro do diretório de um usuário (http://example.com/~username/cakephp/), ou qualquer outra estrutura de URL que já utiliza o mod_rewrite, você irá precisar adicionar instruções RewriteBase para os arquivos .htaccess do CakePHP (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

   Isto pode ser adicionado à mesma seção da diretiva RewriteEngine, por exemplo, o arquivo .htaccess do seu webroot seria algo como:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

Os detalhes dessa mudança vai depender de sua configuração, e pode incluir algumas coisas adicionais que não estão relacionadas ao Cake. Por favor, consulte a documentação online do Apache para mais informações.

URLs amigáveis e Lighttpd

Embora o Lighttpd apresenta um módulo de reescrita, ele não é equivalente ao mod_rewrite do Apache. Para obter "URLs amigáveis" ao usar Lighty você tem duas opções. Uma é usar o mod_rewrite a outra é usar um script LUA com o mod_magnet.

Usando o mod_rewrite O modo mais fácil para se obter URLs amigáveis é adicionando este script na configuração do seu lighty. Basta editar a URL, e tudo deve funcionar. Por favor, note que isto não funciona em instalações do Cake em subdiretórios.

```
$HTTP["host"] =~ "^(www\.)?example.com$" {
        url.rewrite-once = (
                # if the request is for css|files etc, do not pass on to Cake
                "^/(css|files|img|js)/(.*)" => "/$1/$2",
                "^([^\?]*)(\?(.+))?$" => "/index.php?url=$1&$3",
        )
        evhost.path-pattern = "/home/%2-%1/www/www/%4/app/webroot/"
}
```

Usando o mod_magnet Para utiizar URLs amigáveis com o CakePHP e o Lighttpd, coloque este script LUA em /etc/lighttpd/cake.

```
-- little helper function
function file_exists(path)
  local attr = lighty.stat(path)
  if (attr) then
      return true
  else
      return false
  end
end
function removePrefix(str, prefix)
  return str:sub(1,#prefix+1) == prefix.."/" and str:sub(#prefix+2)
end

-- prefix without the trailing slash
local prefix = ''

-- the magic ;)
if (not file_exists(lighty.env["physical.path"])) then
    -- file still missing. pass it to the fastcgi backend
    request_uri = removePrefix(lighty.env["uri.path"], prefix)
    if request_uri then
      lighty.env["uri.path"]          = prefix .. "/index.php"
      local uriquery = lighty.env["uri.query"] or ""
      lighty.env["uri.query"] = uriquery .. (uriquery ~= "" and "&" or "") .. "url=" .
↪. request_uri
      lighty.env["physical.rel-path"] = lighty.env["uri.path"]
      lighty.env["request.orig-uri"]  = lighty.env["request.uri"]
      lighty.env["physical.path"]     = lighty.env["physical.doc-root"] .. lighty.env[
↪"physical.rel-path"]
    end
end
-- fallthrough will put it back into the lighty request loop
```

```
-- that means we get the 304 handling for free. ;)
```

Nota: Se você estiver rodando sua instalação do CakePHP a partir de um subdiretório, você precisa definir o prefix = "subdiretorio" no script acima

Então, informe ao Lighttpd sobre o seu vhost:

```
$HTTP["host"] =~ "example.com" {
        server.error-handler-404  = "/index.php"

        magnet.attract-physical-path-to = ( "/etc/lighttpd/cake.lua" )

        server.document-root = "/var/www/cake-1.2/app/webroot/"

        # Think about getting vim tmp files out of the way too
        url.access-deny = (
                "~", ".inc", ".sh", "sql", ".sql", ".tpl.php",
                ".xtmpl", "Entries", "Repository", "Root",
                ".ctp", "empty"
        )
}
```

URLs amigáveis no nginx

nginx é um servidor popular que, como Lighttpd, usa menos recursos do sistema. O inconveniente é que não faz uso de arquivos .htaccess como o Apache e o Lighttpd, por isso é necessário criar as URLs reescritas na configuração site-available. Dependendo de sua instalação, você terá que modificar isso, mas no mínimo, você irá precisar do PHP sendo executado como FastCGI.

```
server {
    listen   80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com$1 permanent;
}

server {
    listen   80;
    server_name example.com;

    # root directive should be global
    root   /var/www/example.com/public/app/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$uri&$args;
    }

    location ~ \.php$ {
        include /etc/nginx/fastcgi_params;
        try_files $uri =404;
        fastcgi_pass    127.0.0.1:9000;
```

```
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

URL Rewrites no IIS7 (Windows hosts)

O IIS7 não suporta nativamente os arquivos .htaccess. Embora haja add-ons que podem adicionar esse suporte, você também pode importar regras htaccess no IIS para usar as regras de reescritas nativas do CakePHP. Para fazer isso, siga estes passos:

1. Use o *Microsift Web Plataform Installer* para instalar o URL Rewrite Module 2.0.
2. Crie um novo arquivo dentro de sua pasta do CakePHP, chamado web.config.
3. Usando o Notepad ou algum outro editor de XML, copie o seguinte código no seu novo arquivo web.config...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
            <rule name="Imported Rule 1" stopProcessing="true">
            <match url="^(.*)$" ignoreCase="false" />
            <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"␣
↪negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate=
↪"true" />
            </conditions>

            <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true"␣
↪/>

            </rule>

            <rule name="Imported Rule 2" stopProcessing="true">
              <match url="^$" ignoreCase="false" />
              <action type="Rewrite" url="/" />
            </rule>
            <rule name="Imported Rule 3" stopProcessing="true">
              <match url="(.*)" ignoreCase="false" />
              <action type="Rewrite" url="/{R:1}" />
            </rule>
            <rule name="Imported Rule 4" stopProcessing="true">
              <match url="^(.*)$" ignoreCase="false" />
              <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"␣
↪negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate=
↪"true" />
              </conditions>
              <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true
↪" />
            </rule>
            </rules>
        </rewrite>
```

```
    </system.webServer>
</configuration>
```

Também é possível usar a funcionalidade de importação no modulo de reescrita de URL do IIS para importar regras diretamente dos arquivos .htaccess do CakePHP nas pastas /app/, e /app/webroot/ - embora algumas edições no IIS podem ser necessárias para fazê-los funcionar. Importando as regras desta maneira, o IIS irá automaticamente criar o arquivo web.config para você.

Uma vez que o arquivo web.config é criado com o conjunto de regras de reescrita do IIS, links do CakePHP, css, js, e o redirecionamento devem funcionar corretamente.

URL Rewriting

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[11] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Comece agora!

Tudo bem, vamos ver o CakePHP em ação. Dependendo de qual configuração você adotou, você deve apontar seu navegador para http://example.com/ ou http://example.com/cake_install/. Neste ponto, você verá a página padrão do CakePHP e a mensagem do estado da configuração do seu banco de dados.

Parabéns! Você já está pronto para *criar sua primeira aplicação CakePHP*.

Não está funcionando? Se você estiver recebendo erros do PHP relacionados ao fuso horário, descomente uma linha no app/Config/core.php:

```
/**
 * Uncomment this line and correct your server timezone to fix
 * any date & time related errors.
 */
    date_default_timezone_set('UTC');
```

[11] https://github.com/cakephp/docs

CAPÍTULO 3

Visão Geral do CakePHP

Bem vindo ao Cookbook, o manual para o framework de aplicações web CakePHP que torna o desenvolvimento um pedaço de bolo!

Este manual assume que você tenha um entendimento geral sobre PHP e conhecimentos básicos em programação orientada a objetos. Diferentes funcionalidades dentro do framework utilizam diversas tecnologias – como SQL, Javascript e XML – e este manual não tenta explicar estas tecnologias, somente como são usadas neste no contexto.

O que é o CakePHP? Porque usá-lo?

O CakePHP[12] é um framework[13] de desenvolvimento rápido[14] para PHP[15], livre[16] e de Código aberto[17]. Nosso principal objetivo é permitir que você trabalhe de forma estruturada e rápida sem perder a flexibilidade.

O CakePHP tira a monotonia do desenvolvimento web. Nós fornecemos todas as ferramentas que você precisa para começar programando o que realmente deseja: a lógica específica da sua aplicação. Em vez de reinventar a roda a cada vez que se constrói um novo projeto, pegue uma cópia do CakePHP e comece com o interior de sua aplicação.

O CakePHP possui uma equipe de desenvolvedores[18] ativa e uma grande comunidade, trazendo grande valor ao projeto. Além de manter você fora da reinvenção da roda, usando o CakePHP significa que o núcleo da sua aplicação é bem testado e está em constante aperfeiçoamento.

Abaixo segue uma pequena lista dos recursos que você poder desfrutar no CakePHP:

- Comunidade[19] ativa e amigável

[12] https://cakephp.org/
[13] http://pt.wikipedia.org/wiki/Framework
[14] http://pt.wikipedia.org/wiki/Rapid_Application_Development
[15] https://secure.php.net/
[16] http://pt.wikipedia.org/wiki/Licença_MIT
[17] http://pt.wikipedia.org/wiki/Código_aberto
[18] https://github.com/cakephp/cakephp/contributors
[19] https://cakephp.org/feeds

- Licença[20] flexível
- Compatível com o PHP 5.2.6 e superior
- CRUD[21] integrado para interação com o banco de dados
- Scaffolding[22] para criar protótipos
- Geração de código
- Arquitetura MVC[23]
- Requisições feitas com clareza, URLs e rotas customizáveis
- Validações[24] embutidas
- Templates[25] rápidos e flexíveis (Sintaxe PHP, com *helpers*)
- *Helpers* para AJAX, JavaScript, formulários HTML e outros
- Componentes de Email, Cookie, Segurança, Sessão, e Tratamento de Requisições
- Controle de Acessos[26] flexível
- Limpeza dos dados
- Sistema de Cache[27] flexível
- Localização
- Funciona a partir de qualquer diretório do website, com pouca ou nenhuma configuração do Apache

Entendendo o Model-View-Controller

O CakePHP segue o padrão de projeto MVC[28]. Programar usando o MVC separa sua aplicação em três partes principais:

Nota: Optamos por não traduzir as palavras *Model*, *View* e *Controller*. Gostariamos que você se acostumasse com elas pois são muito utilizadas no dia a dia de um desenvolvedor CakePHP. Assim como o Português incorporou diversas palavras estrangeiras, o que você acha de incorporar estas palavras no seu vocabulário?

A camada Model

A camada *Model* (modelo) representa a parte de sua aplicação que implementa a lógica do negócio. Isto significa que ela é responsável por obter os dados convertendo-os em conceitos significativos para sua aplicação, assim como, processar, validar, associar e qualquer outra tarefa relativa ao tratamento dos dados.

À primeira vista, os objetos do tipo Model podem ser vistos como a primeira camada de interação com qualquer banco de dados que você possa estar usando na sua aplicação. Mas em geral eles representam os principais conceitos em torno do qual você implementa sua aplicação.

[20] http://pt.wikipedia.org/wiki/Licença_MIT
[21] http://pt.wikipedia.org/wiki/CRUD
[22] https://en.wikipedia.org/wiki/Scaffold_(programming)
[23] http://pt.wikipedia.org/wiki/MVC
[24] https://en.wikipedia.org/wiki/Data_validation
[25] https://en.wikipedia.org/wiki/Web_template_system
[26] http://pt.wikipedia.org/wiki/Access_Control_List
[27] https://en.wikipedia.org/wiki/Web_cache
[28] http://pt.wikipedia.org/wiki/MVC

No caso de uma rede social, a camada Model cuida de tarefas como as de salvar os dados dos usuários e o relacionamento entre amigos, armazenar e recuperar as fotos dos usuários, encontrar novos amigos para sugestões e etc. Neste exemplo os Models podem ser vistos como «Amigo», «Usuario», «Comentario»e «Foto».

A camada View

Uma *View* exibe uma representação dos dados modelados. Sendo separadas do objeto Model, é responsável por usar as informações disponibilizadas para produzir qualquer interface de apresentação que sua aplicação possa necessitar.

Por exemplo, como a camada Model retorna um conjunto de dados, a view pode usá-los para exibir uma página HTML ou retornar o resultado em um formato XML para que outros o consuma.

A camada View não está limitada à representações dos dados no formato HTML ou texto, podendo ser usada para entregar uma variedade de formatos diferentes, dependendo do que você precisar, como vídeos, músicas, documentos e qualquer outro formato que você puder pensar.

A camada Controller

A camada *Controller* (controlador) lida com as requisições dos usuários. É responsável por retornar uma resposta com a ajuda das camadas Model e View.

Os Controllers podem ser vistos como gerentes tomando os devidos cuidados para que todos os recursos necessários para completar uma tarefa sejam delegados para os trabalhadores corretos. Ele aguarda os pedidos dos clientes, verifica a validade de acordo com as regras de autenticação e autorização, delega dados para serem obtidos ou processados pelos Models e seleciona o tipo correto de apresentação dos dados para o qual o cliente está aceitando para finalmente delegar o trabalho de renderização para a camada de visualização.

Ciclo de Requisição no CakePHP

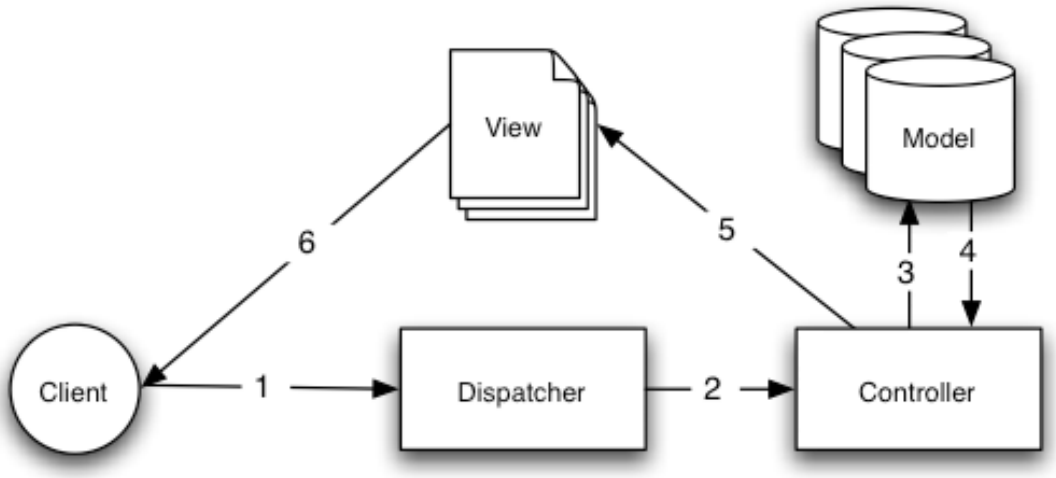

Figura: 1: Uma requisição básica no MVC

Figura 1: Mostra o tratamento de uma simples requisição de um cliente pelo CakePHP.

Um ciclo de requisição típico do CakePHP começa com o usuário solicitando uma página ou recurso em sua aplicação. Esta solicitação é primeiramente processada por um *dispatcher* (expedidor) que irá selecionar o objeto Controller correto para lidar com a solicitação feita.

Assim que a solicitação do cliente chega ao Controller, este irá se comunicar como a camada Model para processar qualquer operação de busca ou armazenamento de dados que for necessário. Após esta comunicação terminar, o Controller continuará delegando, agora para o objeto View correto a tarefa de gerar uma saída resultante dos dados fornecidos pelo Model.

Finalmente quando a saída é gerada, ela é imediatamente enviada para o usuário.

Praticamente todas as requisições feitas para a sua aplicação irão seguir este padrão.

Depois nós iremos adicionar mais alguns detalhes específicos do CakePHP, portanto, tenha isto em mente enquanto prosseguimos.

Benefícios

Por que usar MVC? Porque é um verdadeiro e testado padrão de projeto de software que transforma uma aplicação em pacotes de desenvolvimento rápido, de fácil manutenção e modular. Elaborar tarefas divididas entre models, views e controllers faz com que sua aplicação fique leve. Novas funcionalidades são facilmente adicionadas e pode-se dar nova cara nas características antigas num piscar de olhos. O design modular e separado também permite aos desenvolvedores e designers trabalharem simultaneamente, incluindo a capacidade de se construir um protótipo[29] muito rapidamente. A separação também permite que os desenvolvedores alterem uma parte da aplicação sem afetar outras.

Se você nunca construiu uma aplicação desta forma, leva algum tempo para se acostumar, mas estamos confiantes que uma vez que você tenha construído sua primeira aplicação usando CakePHP, você não vai querer fazer de outra maneira.

Para começar a sua primeira aplicação CakePHP, tente seguir o tutorial para a *construção de um blog*

Onde obter ajuda

Site oficial do CakePHP

https://cakephp.org

O site oficial do CakePHP é sempre o melhor lugar para visitar. Possui links para ferramentas frequentemente utilizadas para desenvolvedores, screencasts, oportunidades de doações e downloads.

O Cookbook

https://book.cakephp.org

Este manual deverá ser provavelmente o primeiro lugar que você vá para obter respostas. Como acontece com muitos outros projetos de código aberto, temos novas pessoas regularmente. Tente primeiro dar o seu melhor para responder às suas próprias perguntas. As respostas podem vir mais lentas, porem, serão duradouras e também aliviará nossa demanda de suporte. Tanto o manual como a API podem ser consultadas online.

O Bakery

https://bakery.cakephp.org

[29] https://en.wikipedia.org/wiki/Software_prototyping

O CakePHP Bakery é um centro de intercâmbio para todas as coisas sobre o CakePHP. Locais para tutoriais, estudos de casos, exemplos de códigos. Uma vez que você estiver familiarizado com o CakePHP, faça logon e compartilhe seus conhecimentos com a comunidade e ganhe fama e fortuna.

A API

https://api.cakephp.org/2.x/

Direto ao ponto e diretamente dos desenvolvedores do núcleo, a API (*Application Programming Interface*) do CakePHP é a mais completa documentação acerca dos mínimos detalhes do funcionamento interno do framework.

Casos de Testes

Se você sentiu que a informação fornecida na API não é suficiente, verifique o código dos casos de testes fornecido com o CakePHP. Eles podem servir como exemplos práticos de utilização das funcionalidades de uma classe:

```
lib/Cake/Test/Case
```

O canal IRC

Canais IRC na rede irc.freenode.net:

- #cakephp – Discussões gerais
- #cakephp-docs – Documentação
- #cakephp-bakery – Bakery
- #cakephp-pt – Discussões gerais em Português

Se você está confuso, dê um grito no canal do CakePHP no IRC. Alguém da equipe de desenvolvimento está geralmente lá, especialmente durante o dia para os usuários da América do Norte e Sul. Adoraríamos ouvir de você, se precisar de alguma ajuda, se quiser encontrar usuários por perto ou quiser doar seu novo carro esportivo.

O Google Group

- http://groups.google.com/group/cakephp-pt

CakePHP também tem um grupo muito ativo no Google. Pode ser um excelente recurso para encontrar respostas arquivadas, perguntas frequentes, e obter respostas para problemas imediatos.

Where to get Help in your Language

French

- French CakePHP Community[30]

[30] http://cakephp-fr.org

CAPÍTULO 4

Controllers

Os controllers correspondem ao "C" no padrão MVC. Após o roteamento ter sido aplicado e o controller correto encontrado, a ação do controller é chamada. Seu controller deve lidar com a interpretação dos dados de uma requisição, certificando-se que os models corretos são chamados e a resposta ou view esperada seja exibida. Os controllers podem ser vistos como intermediários entre a camada Model e View. Você vai querer manter seus controllers magros e seus Models gordos. Isso lhe ajudará a reutilizar seu código e testá-los mais facilmente.

Mais comumente, controllers são usados para gerenciar a lógica de um único model. Por exemplo, se você está construindo um site para uma padaria online, você pode ter um `RecipesController` e um `IngredientsController` gerenciando suas receitas e seus ingredientes. No CakePHP, controllers são nomeados de acordo com o model que manipulam. É também absolutamente possível ter controllers que usam mais de um model.

Os controllers da sua aplicação são classes que estendem a classe CakePHP `AppController`, a qual por sua vez estende a classe *`Controller`* do CakePHP. A classe `AppController` pode ser definida em `/app/Controller/AppController.php` e deve conter métodos que são compartilhados entre todos os seus controllers.

Os controllers fornecem uma série de métodos que são chamados de ações. Ações são métodos em um controller que manipulam requisições. Por padrão, todos os métodos públicos em um controller são ações e acessíveis por urls.

A Classe AppController

Como mencionado anteriormente, a classe `AppController` é a mãe de todos os outros controllers da sua aplicação. O próprio `AppController` é estendida da classe `Controller` que faz parte da biblioteca do CakePHP. Assim sendo, `AppController` é definido em `/app/Controller/AppController.php` como:

```
class AppController extends Controller {
}
```

Os atributos e métodos criados em `AppController` vão estar disponíveis para todos os controllers da sua aplicação. Este é o lugar ideal para criar códigos que são comuns para todos os seus controllers. Componentes (que você vai

aprender mais tarde) são a melhor alternativa para códigos que são usados por muitos (mas não obrigatoriamente em todos) controllers.

Enquanto regras normais de herança de classes orientadas à objetos são aplicadas, o CakePHP também faz um pequeno trabalho extra quando se trata de atributos especiais do controller. A lista de componentes (components) e helpers usados no controller são tratados diferentemente. Nestes casos, as cadeias de valores do `AppController` são mescladas com os valores de seus controllers filhos. Os valores dos controllers filhos sempre sobrescreveram os do `AppController`.

Nota: O CakePHP mescla as seguintes variáveis do `AppController` em controllers da sua aplicação:

- $components
- $helpers
- $uses

Lembre-se de adicionar os helpers Html e Form padrões se você incluiu o atributo `$helpers` em seu `AppController`.

Também lembre de fazer as chamadas de callbacks do `AppController` nos controllers filhos para obter melhores resultados:

```
function beforeFilter() {
    parent::beforeFilter();
}
```

Parâmetros de Requisição

Quando uma requisição é feita para uma aplicação CakePHP, a classe `Router` e a classe `Dispatcher` do Cake usa a *Routes Configuration* para encontrar e criar o controller correto. Os dados da requisição são encapsulados em um objeto de requisição. O CakePHP coloca todas as informações importantes de uma requisição na propriedade `$this->request`. Veja a seção *Objetos de Requisição e Resposta* para mais informações sobre o objeto de requisição do CakePHP.

Ações de Controllers

Retornando ao nosso exemplo da padaria online, nosso controller `RecipesController` poderia conter as ações `view()`, `share()` e `search()` e poderia ser encontrado em `/app/Controller/RecipesController.php` contendo o código a seguir:

```
# /app/Controller/RecipesController.php

class RecipesController extends AppController {
    function view($id) {
        // a lógica da ação vai aqui
    }

    function share($customer_id, $recipe_id) {
        // a lógica da ação vai aqui
    }

    function search($query) {
```

```
        // a lógica da ação vai aqui
    }
}
```

Para que você use de forma eficaz os controllers em sua aplicação, nós iremos cobrir alguns dos atributos e métodos inclusos no controller fornecido pelo CakePHP.

Ciclo de Vida dos Callbacks em uma Requisição

class Controller

Os controllers do CakePHP vêm equipados com callbacks que você pode usar para inserir lógicas em torno do ciclo de vida de uma requisição:

`Controller::beforeFilter()`
: Este método é executado antes de cada ação dos controllers. É um ótimo lugar para verificar se há uma sessão ativa ou inspecionar as permissões de um usuário.

> **Nota:** O método beforeFilter() será chamado para ações não encontradas e ações criadas pelo scaffold do Cake.

`Controller::beforeRender()`
: Chamada após a lógica da ação de um controller, mas antes da view ser renderizada. Este callback não é usado com frequência mas pode ser preciso se você chamar o método `render()` manualmente antes do término de uma ação.

`Controller::afterFilter()`
: Chamada após cada ação dos controllers, e após a completa renderização da view. Este é o último método executado do controller.

Em adição aos callbacks dos controllers, os *Componentes* também fornecem um conjunto de callbacks similares.

Métodos dos Controllers

Para uma lista completa dos métodos e suas descrições, visite a API do CakePHP. Siga para https://api.cakephp.org[31].

Interagindo Com as Views

Os controllers interagem com as views de diversas maneiras. Primeiramente eles são capazes de passar dados para as views usando o método `set()`. Você também pode no seu controller decidir qual classe de View usar e qual arquivo deve ser renderizado.

`Controller::set`(*string $var*, *mixed $value*)
: O método `set()` é a principal maneira de enviar dados do seu controller para a sua view. Após ter usado o método `set()`, a variável pode ser acessada em sua view:

```
// Primeiro você passa os dados do controller:

$this->set('color', 'pink');

//Então, na view, você pode utilizar os dados:
```

[31] https://api.cakephp.org/2.x/class-Controller.html

```
?>

Você selecionou a cobertura <?php echo $color; ?> para o bolo.
```

O método `set()` também aceita um array associativo como primeiro parâmetro, podendo oferecer uma forma rápida para atribuir uma série de informações para a view.

Alterado na versão 1.3: Chaves de arrays não serão mais flexionados antes de serem atribuídas à view ("underscored_key" não se torna "underscoredKey", etc.):

```
$data = array(
    'color' => 'pink',
    'type' => 'sugar',
    'base_price' => 23.95
);

// Torna $color, $type e $base_price
// disponível na view:

$this->set($data);
```

O atributo `$pageTitle` não existe mais, use o método `set()` para definir o título na view:

```
$this->set('title_for_layout', 'This is the page title');
?>
```

`Controller::render` (*string $action*, *string $layout*, *string $file*)

O método `render()` é chamado automaticamente no fim de cada ação requisitada de um controller. Este método executa toda a lógica da view (usando os dados que você passou usando o método `set()`), coloca a view dentro do seu layout e serve de volta para o usuário final.

O arquivo view usado pelo método `render()` é determinado por convenção. Se a ação `search()` do controller `RecipesController` é requisitada, o arquivo view encontrado em `/app/View/Recipes/search.ctp` será renderizado:

```
class RecipesController extends AppController {
...
    function search() {
        // Renderiza a view em /View/Recipes/search.ctp
        $this->render();
    }
...
}
```

Embora o CakePHP irá chamar o método `render` automaticamente (ao menos que você altere o atributo `$this->autoRender` para `false`) após cada ação, você pode usá-lo para especificar um arquivo view alternativo alterando o nome de ação no controller usando o parâmetro `$action`.

Se o parâmetro `$action` começar com `'/'` é assumido que o arquivo view ou elemento que você queira usar é relativo ao diretório `/app/View`. Isto permite a renderização direta de elementos, muito útil em chamadas Ajax.

```
// Renderiza o elemento presente em /View/Elements/ajaxreturn.ctp
$this->render('/Elements/ajaxreturn');
```

Você também pode especificar uma arquivo view ou elemento usando o terceiro parâmetro chamado `$file`. O parâmetro `$layout` permite você especificar o layout em que a view será inserido.

Renderizando Uma View Específica

Em seu controller você pode querer renderizar uma view diferente do que a convenção proporciona automaticamente. Você pode fazer isso chamando o método `render()` diretamente. Após ter chamado o método `render()`, o CakePHP não irá tentar renderizar novamente a view:

```
class PostsController extends AppController {
    function my_action() {
        $this->render('custom_file');
    }
}
```

Isto irá renderizar o arquivo `app/View/Posts/custom_file.ctp` ao invés de `app/View/Posts/my_action.ctp`

Controle de Fluxo

`Controller::redirect` (*mixed $url*, *integer $status*, *boolean $exit*)

O método de controle de fluxo que você vai usar na maioria das vezes é o `redirect()`. Este método recebe seu primeiro parâmetro na forma de uma URL relativa do CakePHP. Quando um usuário executou um pedido que altera dados no servidor, você pode querer redirecioná-lo para uma outra tela de recepção.:

```
function place_order() {
    // Logic for finalizing order goes here
    if ($success) {
        $this->redirect(array('controller' => 'orders', 'action' => 'thanks'));
    } else {
        $this->redirect(array('controller' => 'orders', 'action' => 'confirm'));
    }
}
```

Nota: Você pode aprender mais sobre a importância de redirecionar o usuário após um formulário do tipo POST no artigo Post/Redirect/Get (en)[32].

Você também pode usar uma URL relativa ou absoluta como argumento:

```
$this->redirect('/orders/thanks'));
$this->redirect('http://www.example.com');
```

Você também pode passar dados para a ação:

```
// observe o parâmetro $id
$this->redirect(array('action' => 'edit', $id));
```

O segundo parâmetro passado no `redirect()` permite você definir um código de status HTTP para acompanhar o redirecionamento. Você pode querer usar o código 301 (movido permanentemente) ou 303 (siga outro), dependendo da natureza do redirecionamento.

O método vai assumir um `exit()` após o redirecionamento ao menos que você passe o terceiro parâmetro como `false`.

Se você precisa redirecionar o usuário de volta para a página que fez a requisição, você pode usar:

[32] https://en.wikipedia.org/wiki/Post/Redirect/Get

```
$this->redirect($this->referer());
```

`Controller::flash`(*string $message*, *string $url*, *integer $pause*, *string $layout*)
Assim como o método `redirect()`, o método `flash()` é usado para direcionar o usuário para uma nova página após uma operação. O método `flash()` é diferente na forma de transição, mostrando uma mensagem antes de transferir o usuário para a URL especificada.

O primeiro parâmetro deve conter a mensagem que será exibida e o segundo parâmetro uma URL relativa do CakePHP. O Cake irá mostrar o conteúdo da variável `$message` pelos segundos especificados em `$pause` antes de encaminhar o usuário para a URL especificada em `$url`.

Se existir um template particular que você queira usar para mostrar a mensagem para o usuário, você deve especificar o nome deste layout passando o parâmetro `$layout`.

Para mensagens flash exibidas dentro de páginas, de uma olhada no método `setFlash()` do componente `SessionComponent`.

Callbacks

Em adição ao *Ciclo de Vida dos Callbacks em uma Requisição*. O CakePHP também suporta callbacks relacionados a scaffolding.

`Controller::beforeScaffold`(*$method*)
`$method` é o nome do método chamado, por exemplo: index, edit, etc.

`Controller::scaffoldError`(*$method*)
`$method` é o nome do método chamado, por exemplo: index, edit, etc.

`Controller::afterScaffoldSave`(*$method*)
`$method` é o nome do método chamado, podendo ser: edit ou update.

`Controller::afterScaffoldSaveError`(*$method*)
`$method` é o nome do método chamado, podendo ser: edit ou update.

Outros Métodos Úteis

`Controller::constructClasses`()
Este método carrega os models requeridos pelo controller. Este processo de carregamento é feito normalmente pelo CakePHP, mas pode ser útil quando for acessar controllers de outras perspectivas. Se você precisa de um controller num script de linha de comando ou para outros lugares, `constructClasses()` pode vir a calhar.

`Controller::referer`(*mixed $default = null*, *boolean $local = false*)
Retorna a URL de referência para a requisição atual. O parâmetro `$default` pode ser usado para fornecer uma URL padrão a ser usada caso o HTTP_REFERER não puder ser lido do cabeçalho da requisição. Então, ao invés de fazer isto:

```
class UserController extends AppController {
    function delete($id) {
        // delete code goes here, and then...
        if ($this->referer() != '/') {
            $this->redirect($this->referer());
        } else {
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

Você pode fazer isto:

```
class UserController extends AppController {
    function delete($id) {
        // delete code goes here, and then...
        $this->redirect($this->referer(array('action' => 'index')));
    }
}
```

Se o parâmetro `$default` não for passado, o comportamento padrão é a raiz do seu domínio - `'/'`.

Se o parâmetro `$local` for passado como `true`, o redirecionamento restringe a URL de referência apenas para o servidor local.

`Controller::disableCache()`
Usado para dizer ao **browser** do usuário não fazer cache do resultado exibido. Isto é diferente do cache de views, abordado em um capítulo posterior.

O cabeçalho enviado para este efeito são:

```
Expires: Mon, 26 Jul 1997 05:00:00 GMT
Last-Modified: [data e hora atual] GMT
Cache-Control: no-store, no-cache, must-revalidate
Cache-Control: post-check=0, pre-check=0
Pragma: no-cache
```

`Controller::postConditions` (*array $data*, *mixed $op*, *string $bool*, *boolean $exclusive*)
Use este método para transformar um conjunto de dados POSTados de um model (vindos de inputs compatíveis com o FormHelper) em um conjunto de condições de busca. Este método oferece um atalho rápido no momento de construir operações de busca. Por exemplo, um usuário administrativo pode querer pesquisar pedidos para saber quais itens precisarão ser enviados. Você pode usar o `FormHelper` do CakePHP para criar um formulário de busca para o model Order. Assim, uma ação de um controller pode usar os dados enviados deste formulário e criar as condições de busca necessárias para completar a tarefa:

```
function index() {
    $conditions = $this->postConditions($this->request->data);
    $orders = $this->Order->find('all', compact('conditions'));
    $this->set('orders', $orders);
}
```

Se `$this->request->data['Order']['destination']` for igual a «Old Towne Bakery», o método `postConditions()` converte esta condição em um array compatível para o uso em um método Model->find(). Neste caso, `array('Order.destination' => 'Old Towne Bakery')`.

Se você quiser usar um operador diferente entre os termos, informe-os usando o segundo parâmetro:

```
/*
Conteúdo do atributo $this->request->data
array(
    'Order' => array(
        'num_items' => '4',
        'referrer' => 'Ye Olde'
    )
)
*/

// Vamos pegar os pedidos que possuem no mínimo 4 itens e que contém 'Ye Olde'
$conditions = $this->postConditions(
    $this->request->data,
```

```
    array(
        'num_items' => '>=',
        'referrer' => 'LIKE'
    )
);
$orders = $this->Order->find('all', compact('conditions'));
```

O terceiro parâmetro permite você dizer ao CakePHP qual operador SQL booleano usar entre as condições de busca. Strings como "AND", "OR" e "XOR" são todos valores válidos.

Finalmente, se o último parâmetro for passado como `true`, e a variável `$op` for um array, os campos não inclusos em `$op` não serão retornados entre as condições.

`Controller::paginate()`
Este método é usado para fazer a paginação dos resultados retornados por seus models. Você pode especificar o tamanho da página (quantos resultados serão retornados), as condições de busca e outros parâmetros. Veja a seção pagination para mais detalhes sobre como usar o método `paginate()`

`Controller::requestAction` (*string $url*, *array $options*)
Este método chama uma ação de um controller de qualquer lugar e retorna os dados da ação requisitada. A `$url` passada é uma URL relativa do Cake (/controller_name/action_name/params). Para passar dados extras para serem recebidos pela ação do controller, adicione-os no parâmetro `options` em um formato de array.

Nota: Você pode usar o `requestAction()` para recuperar uma view totalmente renderizada passando `'return'` no array de opções: `requestAction($url, array('return'));`. É importante notar que fazendo uma requisição usando "return" em um controller podem fazer com que tags javascripts e css não funcionem corretamente.

Aviso: Se o método `requestAction()` for usado sem fazer cache apropriado do resultado obtido, a performance da ação pode ser bem ruim. É raro o uso apropriado deste método em um controller ou model.

O uso do `requestAction` é melhor usado em conjunto com elementos (cacheados) como uma maneira de recuperar dados para um elemento antes de renderizá-los. Vamos usar o exemplo de por o elemento «últimos comentários» no layout. Primeiro nós precisamos criar um método no controller que irá retornar os dados:

```
// Controller/CommentsController.php
class CommentsController extends AppController {
    function latest() {
        return $this->Comment->find('all', array('order' => 'Comment.created DESC
↪', 'limit' => 10));
    }
}
```

Se agora nós criarmos um elemento simples para chamar este método:

```
// View/Elements/latest_comments.ctp

$comments = $this->requestAction('/comments/latest');
foreach ($comments as $comment) {
    echo $comment['Comment']['title'];
}
```

Nós podemos por este elemento em qualquer lugar para ter a saída usando:

```
echo $this->element('latest_comments');
```

Fazendo desta maneira, sempre que o elemento for renderizado, uma requisição será feita para nosso controller para pegar os dados, processá-los e retorná-los. Porém, de acordo com o aviso acima, é melhor fazer uso de caching do elemento para evitar um processamento desnecessário. Modificando a chamada do elemento para se parecer com isto:

```
echo $this->element('latest_comments', array('cache' => '+1 hour'));
```

A chamada para o `requestAction` não será feita enquanto o arquivo de cache do elemento existir e for válido.

Além disso, o `requestAction` pode receber uma URL no formato de array do Cake:

```
echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'featured'),
    array('return')
);
```

Isto permite o `requestAction` contornar o uso do método `Router::url()` para descobrir o controller e a ação, aumentando a performance. As URLs baseadas em arrays são as mesmas usadas pelo método `HtmlHelper::link()` com uma diferença, se você está usando parâmetros nomeados ou passados, você deve colocá-los em um segundo array envolvendo elas com a chave correta. Isto deve ser feito porque o `requestAction` mescla o array de parâmetros nomeados (no segundo parâmetro do requestAction) com o array `Controller::params` e não coloca explicitamente o array de parâmetros nomeados na chave "named". Além disso, membros do array `$options` serão disponibilizados no array `Controller::params` da ação que for chamada.:

```
echo $this->requestAction('/articles/featured/limit:3');
echo $this->requestAction('/articles/view/5');
```

Um array no `requestAction` poderia ser:

```
echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'featured'),
    array('named' => array('limit' => 3))
);

echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'view'),
    array('pass' => array(5))
);
```

Nota: Diferente de outros lugares onde URLs no formato de arrays são análogas as URLs no formato de string, o `requestAction` tratam elas diferentemente.

Quando for usar URLs no formato de arrays em conjunto com o `requestAction()` você deve especificar **todos** os parâmetros que você vai precisar na requisição da ação. Isto inclui parâmetros como `$this->request->data`. Além disso, todos os parâmetros requeridos, parâmetros nomeados e «passados» devem ser feitos no segundo array como visto acima.

`Controller::loadModel` (*string $modelClass*, *mixed $id*)

O método `loadModel` vem a calhar quando você precisa usar um model que não é padrão do controller ou o seu model não está associado com este.

```
$this->loadModel('Article');
$recentArticles = $this->Article->find('all', array('limit' => 5, 'order' =>
↪'Article.created DESC'));

$this->loadModel('User', 2);
$user = $this->User->read();
```

Atributos do Controller

Para uma completa lista dos atributos dos controllers e suas descrições, visite a API do CakePHP. Siga para https://api.cakephp.org/2.x/class-Controller.html.

property Controller::$**name**

O atributo `$name` deve ser definido com o nome do controller. Normalmente é apenas a forma plural do nome do model principal que o controller usa. Esta propriedade não é requerida mas salva o CakePHP de ter que flexionar o nome do model para chegar no valor correto:

```
# Exemplo de uso do atributo $name do controller

class RecipesController extends AppController {
   public $name = 'Recipes';
}
```

$components, $helpers e $uses

Os seguintes atributos do controller usados com mais frequência dizem ao CakePHP quais helpers, componentes e models você irá usar em conjunto com o controller corrente. Usar estes atributos faz com que as classes MVC dadas por `$components` e `$uses` sejam disponibilizadas como atributos no controller (por exemplo, `$this->ModelName`) e os dados por `$helpers` disponibilizados como referências para objetos apropriados (`$this->{$helpername}`) na view.

Nota: Cada controller possui algumas destas classes disponibilizadas por padrão, então você pode nem ao menos precisar de configurar estes atributos.

---

property Controller::$**uses**

Os controllers possuem acesso ao seu model principal por padrão. Nosso controller Recipes terá a classe model Recipe disposta em `$this->Recipe` e nosso controller Products também apresenta o model Product em `$this->Product`. Porém, quando for permitir um controller acessar models adicionais pela configuração do atributo `$uses`, o nome do model do controller atual deve também estar incluso. Este caso é ilustrado no exemplo logo abaixo.

Se você não quiser usar um model em seu controller, defina o atributo como um array vazio (`public $uses = array()`). Isto lhe permitirá usar um controller sem a necessidade de um arquivo model correspondente.

property Controller::$**helpers**

Os helpers *Html*, *Form* e *Session* são disponibilizados por padrão como é feito o `SessionComponent`. Mas se você escolher definir seu próprio array de `$helpers` no `AppController`, tenha certeza de incluir o `Html` e o `Form` se quiser que eles continuem a estar disponíveis nos seus controllers. Você também pode passar configurações na declaração de seus helpers. Para aprender mais sobre estas classes, visite suas respectivas seções mais tarde neste manual.

Vamos ver como dizer para um controller do Cake que você planeja usar classes MVC adicionais:

```
class RecipesController extends AppController {
    public $uses = array('Recipe', 'User');
    public $helpers = array('Js');
    public $components = array('RequestHandler');
}
```

Cada uma destas variáveis são mescladas com seus valores herdados, portanto, não é necessário (por exemplo) redeclarar o `FormHelper` ou qualquer uma das classes que já foram declaradas no seu `AppController`.

property `Controller::$`**components**
O array de componentes permite que você diga ao CakePHP quais *Componentes* um controller irá usar. Como o `$helpers` e o `$uses`, `$components` são mesclados com os definidos no `AppController`. E assim como nos `$helpers`, você pode passar configurações para os componentes. Veja *Configurando Componentes* para mais informações.

Outros Atributos

Enquanto você pode conferir todos os detalhes de todos os atributos dos controllers na API, existem outros atributos dos controllers que merecem suas próprias seções neste manual.

property `Controller::$`**cacheAction**
O atributo `$cacheAction` é usado para definir a duração e outras informações sobre o cache completo de páginas. Você pode ler mais sobre o *caching* completo de páginas na documentação do `CacheHelper`.

property `Controller::$`**paginate**
O atributo `$paginate` é uma propriedade de compatibilidade obsoleta. Usando este atributo, o componente `PaginatorComponent` será carregado e configurado, no entanto, é recomendado atualizar seu código para usar as configurações normais de componentes:

```
class ArticlesController extends AppController {
    public $components = array(
        'Paginator' => array(
            'Article' => array(
                'conditions' => array('published' => 1)
            )
        )
    );
}
```

Mais sobre Controllers

Objetos de Requisição e Resposta

Os objetos *Request* e *Response* são novos no CakePHP 2.0. Anteriormente, estes objetos eram representados por arrays e os métodos relacionados espalhados nas classes `RequestHandlerComponent`, `Router`, `Dispatcher` e *`Controller`*. Não havia nenhum objeto com autoridades sobre as informações contidas em uma requisição. No CakePHP 2.0, as classes *`CakeRequest`* e *`CakeResponse`* são usadas para este propósito.

CakeRequest

A classe *`CakeRequest`* é o objeto padrão para requisições usadas no CakePHP. Ela centraliza inúmeras funcionalidades para interagir com os dados das requisições.

A cada requisição feita, um `CakeRequest` é criado e passado por referência para as várias camadas de uma aplicação que usam os dados de uma requisição. Por padrão, `CakeRequest` é atribuído em `$this->request` e é disponibilizado nos controller, views e helpers. Você pode também acessá-la em componentes usando a referência do controller. Algumas das tarefas que o `CakeRequest` executa inclui:

- Processar os arrays GET, POST e FILES na estrutura de dados que você está familiarizado.
- Fornecer a introspecção do ambiente pertencente a requisição. Coisas como cabeçalhos enviados, endereço IP dos clientes e informações de domínio/subdomínio sobre o servidor que a aplicação está rodando.
- Prover acesso aos parâmetros da requisição pelo uso de arrays ou propriedades do objeto.

Acessando parâmetros de uma Requisição

O CakeRequest expõe várias maneiras de acessar os parâmetros de uma requisição. A primeira é o acesso por índices de array, a segunda maneira é pelo `$this->request->params` e a terceira por propriedades do objeto:

```
$this->request['controller'];
$this->request->controller;
$this->request->params['controller']
```

Todas as alternativas acima irão acessar o mesmo valor. Foram criadas várias maneiras de acessar os parâmetros para facilitar a migração de aplicações existentes que utilizam versões antigas do Cake. Todos os *Route Elements* podem ser acessados por esta interface.

além dos *Route Elements*, muitas vezes você precisará ter acesso aos *Passed Arguments* e os *Named Parameters*. Ambos estarão disponíveis no objeto da classe CakeRequest:

```
// Argumentos passados
$this->request['pass'];
$this->request->pass;
$this->request->params['pass'];

// Parâmetros nomeados
$this->request['named'];
$this->request->named;
$this->request->params['named'];
```

Todos irão lhe proporcionar o acesso aos argumentos passados e os parâmetros nomeados. Existem diversos parâmetros que são importantes/úteis que o CakePHP utiliza internamente e podem também ser encontrados nos parâmetros da requisição:

- `plugin` O nome do plugin que trata a requisição. será `null` quando não for nenhum plugin.
- `controller` O nome do controller que trata a requisição corrente.
- `action` A ação responsável por manipular a requisição corrente.
- `prefix` O prefixo da ação corrente. Veja *Prefix Routing* para mais informações.
- `bare` Presente quando uma requisição chega por meio do método `requestAction()` e inclui a opção `bare`. Requisições despidas (bare) não possuem layouts.
- `requested` Presente e definida como `true` quando vindas de um uma chamada do método `requestAction()`.

Acessando parâmetros do tipo querystring

Parâmetros do tipo «query string» presentes tipicamente em requisições do tipo GET podem ser lidos usando *CakeRequest::$query*:

```
// Sendo a url /posts/index?page=1&sort=title
$this->request->query['page'];

// Você também pode acessar o valor via array
$this->request['url']['page'];
```

Acessando dados em requisições do tipo POST

Todos os dados encontrados em requisições do tipo POST podem ser acessados usando o atributo *CakeRequest::$data*. Qualquer dado passado por formulários que contenha o prefixo `data` terá este prefixo removido. Por exemplo:

```
// Uma tag input com o atributo "name" igual a 'data[Post][title]' é
acessavel em:

$this->request->data['Post']['title'];
```

você pode acessar a propriedade data como também pode usar o método *CakeRequest::data()* para ler os dados do array de forma a evitar erros. Qualquer chave que não exista irá retornar o valor `null`. Desta maneira não é preciso verificar se a chave existe antes de usá-la:

```
$foo = $this->request->data('Valor.que.nao.existe');
// $foo == null
```

Acessando dados XML ou JSON

Aplicações que empregam métodos *REST* muitas vezes transferem dados em formatos não codificados no padrão URL. Você pode ler estas entradas de dados com qualquer formato usando o método *CakeRequest::input()*. Fornecendo uma função de decodificação, você pode receber o conteúdo em um formato desserializado:

```
// Obtém dados codificados no formato JSON submetidos por um método PUT/POST
$data = $this->request->input('json_decode');
```

Como alguns métodos de desserialização requerem parâmetros adicionais ao serem chamados, como a opção «as array» da função `json_decode` ou se você quiser um XML convertido em um objeto `DOMDocument`, o método *CakeRequest::input()* também suporta a passagem de parâmetros adicionais:

```
// Obtém dados codificados em XML submetidos por um método PUT/POST
$data = $this->request->input('Xml::build', array('return' => 'domdocument'));
```

Acessando informações sobre o caminho das URLs

O CakeRequest também fornece informações úteis sobre o caminho de sua aplicação. O *CakeRequest::$base* e o *CakeRequest::$webroot* são úteis para gerar urls e determinar se sua aplicação está ou não em um subdiretório.

Inspecionando a Requisição

Anteriormente, era preciso utilizar o `RequestHandlerComponent` para detectar vários aspectos de uma requisição. Estes métodos foram transferidos para o `CakeRequest` e esta classe oferece uma nova interface enquanto mantem certa compatibilidade com as versões anteriores do Cake:

```
$this->request->is('post');
$this->request->isPost();
```

Ambas os métodos chamados irão retornar o mesmo valor. Por enquanto os métodos ainda são disponibilizados no RequestHandler mas são depreciados e ainda podem ser removidos futuramente. Você também pode facilmente estender os detectores que estão disponíveis usando o método *`CakeRequest::addDetector()`* para criar novos tipos de detectores. Existem quatro formas diferentes de detectores que você pode criar:

- Comparação de valores de ambiente - Uma comparação feita em valores do ambiente compara valores encontrados pela função `env()` no ambiente da aplicação, com o valor fornecido.
- Comparação por expressão regular - Permite comparar valores encontrados pela função `env()` com uma expressão regular fornecida.
- Comparação baseada em opções - Usa uma lista de opções para criar expressões regulares. Chamadas subsequentes para adicionar opções já fornecidas ao detector serão mescladas.
- Detectores do tipo Callback - Permitem fornecer um «callback» para tratar a verificação. O callback irá receber o objeto de requisição como parâmetro único.

Alguns exemplos de uso:

```
// Adiciona um detector baseado em variáveis do ambiente
$this->request->addDetector('post', array('env' => 'REQUEST_METHOD', 'value' => 'POST
↪'));

// Adicionar um detector usando expressões regulares
$this->request->addDetector('iphone', array('env' => 'HTTP_USER_AGENT', 'pattern' =>
↪'/iPhone/i'));

// Adicionar um detector baseado em uma lista de opções
$this->request->addDetector('internalIp', array(
    'env' => 'CLIENT_IP',
    'options' => array('192.168.0.101, '192.168.0.100')
));

// Adiciona um detector callback. Pode ser tanto uma função anônima
// quanto o nome de uma função a ser chamada.
$this->request->addDetector('awesome', function ($request) {
    return isset($request->awesome);
});
```

O `CakeRequest` também inclui métodos como *`CakeRequest::domain()`*, *`CakeRequest::subdomains()`* e *`CakeRequest::host()`* para ajudar em aplicações que utilizam subdomínios, tornando a vida um pouco mais fácil.

Existem vários detectores inclusos no Cake que você já pode usar:

- `is('get')` Verifica se a requisição corrente é do tipo GET.
- `is('put')` Verifica se a requisição corrente é do tipo PUT.
- `is('post')` Verifica se a requisição corrente é do tipo POST.
- `is('delete')` Verifica se a requisição corrente é do tipo DELETE.

- `is('head')` Verifica se a requisição corrente é do tipo HEAD.
- `is('options')` Verifica se a requisição corrente é do tipo OPTIONS.
- `is('ajax')` Verifica se a requisição corrente acompanha o cabeçalho X-Requested-With = XMLHttpRequest.
- `is('ssl')` Verifica se a requisição corrente é via SSL.
- `is('flash')` Verifica se a requisição foi feita por um objeto do Flash.
- `is('mobile')` Verifica se a requisição veio de uma lista comum de dispositivos móveis.

CakeRequest e o RequestHandlerComponent

Como muitas das caracteristicas que o `CakeRequest` oferece eram de domínio do componente `RequestHandlerComponent`, foi preciso repensar como esta se encaixa no quadro atual. Para o CakePHP 2.0, a classe `RequestHandlerComponent` age como uma cereja em cima do bolo. Provendo uma camada adicional de funcionalidades sobre o `CakeRequest`, como a mudança do layout baseado no tipo de conteúdo ou chamadas em ajax. A separação destas duas classes permitem você escolher mais facilmente o que você quer e precisa.

Interagindo com outros aspectos da requisição

Você pode usar o *CakeRequest* para introspectar uma variedade de coisas sobre a requisição. Além dos detectores, você também pode encontrar outras informações vindas de várias propriedades e métodos.

- `$this->request->webroot` contém o diretório webroot (a raiz do diretório web).
- `$this->request->base` contém o caminho base.
- `$this->request->here` contém a uri solicitada da requisição corrente.
- `$this->request->query` contém os parâmetros enviados por «query strings».

API do CakeRequest

class CakeRequest
A classe CakeRequest encapsula o tratamento e introspecção dos parâmetros das requisições.

CakeRequest::**domain**()
Retorna o nome do domínio onde sua aplicação esta sendo executada.

CakeRequest::**subdomains**()
Retorna os subdomínios de onde sua aplicação está sendo executada em um formato de array.

CakeRequest::**host**()
Retorna o host em que sua aplicação esta sendo executada.

CakeRequest::**method**()
Retorna o método HTTP em que a requisição foi feita.

CakeRequest::**referer**()
Retorna o endereço que referenciou a requisição.

CakeRequest::**clientIp**()
Retorna o endereço IP do visitante corrente.

CakeRequest::**header**()
Permite você acessar qualquer cabeçalho `HTTP_*` que tenha sido usado na requisição:

```
$this->request->header('User-Agent');
```

Retornaria o «user agent» utilizado para a solicitação.

`CakeRequest::input`(*$callback*[, *$options*])
Resgata os dados de entrada de uma requisição. Opcionalmente o resultado é passado por uma função de decodificação dos dados. Parâmetros adicionais para a função de decodificação podem ser passadas como argumentos para `input()`.

`CakeRequest::data`(*$key*)
Fornece acesso aos dados da requisição numa notação pontuada, permitindo a leitura e modificação dos dados da requisição. Chamadas também podem ser encadeadas:

```
// Modifica alguns dados da requisição, assim você pode popular
// previamente alguns campos dos formulários.
$this->request->data('Post.title', 'New post')
    ->data('Comment.1.author', 'Mark');

// Você também pode ler os dados.
$value = $this->request->data('Post.title');
```

`CakeRequest::is`(*$check*)
Verifica se uma requisição corresponde a um certo critério. Utiliza os detectores inclusos por padrão além das regras adicionadas com o método *`CakeRequest::addDetector()`*.

`CakeRequest::addDetector`(*$name*, *$callback*)
Adiciona um detector para ser usado com o método `is()`. Veja *Inspecionando a Requisição* para mais informações.

`CakeRequest::accepts`(*$type*)
Descobre quais os tipos de conteúdo que o cliente aceita ou verifica se ele aceita um determinado tipo de conteúdo.

Obtém todos os tipos:

```
<?php
$this->request->accepts();
```

Verifica apenas um tipo:

```
$this->request->accepts('application/json');
```

static `CakeRequest::acceptLanguage`(*$language*)
Obter todas os idiomas aceitos pelo cliente ou verifica se um determinado idioma é aceito.

Obtém uma lista dos idiomas aceitos:

```
CakeRequest::acceptLanguage();
```

Verifica se um idioma específico é aceito:

```
CakeRequest::acceptLanguage('es-es');
```

property `CakeRequest::$data`
Um array de dados enviados pelo método POST. Você pode usar o método *`CakeRequest::data()`* para ler o conteúdo desta propriedade de uma forma a suprimir avisos quando a chave informada não existir.

property `CakeRequest::$query`
Um array de parâmetros passados por «query strings».

property CakeRequest::$**params**
Um array contendo os elementos da rota e os parâmetros da requisição.

property CakeRequest::$**here**
Contém a uri solicitada no momento da requisição.

property CakeRequest::$**base**
O caminho de base para a aplicação, geralmente equivale a /, ao menos que sua aplicação esteja em um subdiretório.

property CakeRequest::$**webroot**
O diretório web de sua aplicação.

CakeResponse

O *CakeResponse* é a classe padrão para respostas no CakePHP. Ela encapsula inúmeras características e funcionalidades para gerar respostas HTTP em sua aplicação. Ela também auxilia nos testes da aplicação e pode ser «forjada», permitindo inspecionar os cabeçalhos que serão enviados. Como na classe *CakeRequest*, o *CakeResponse* consolida vários métodos encontrados previamente no *Controller*, RequestHandlerComponent e Dispatcher. Os métodos antigos foram depreciados, favorecendo o uso do *CakeResponse*.

CakeResponse fornece uma interface para envolver as tarefas comuns relacionadas ao envio de respostas para o cliente como:

- Enviar cabeçalhos de redirecionamento.
- Enviar cabeçalhos com o tipo de conteúdo.
- Enviar qualquer outro cabeçalho.
- Enviar o corpo da resposta.

Alterando a classe de Resposta

O CakePHP utiliza o CakeResponse por padrão. O CakeResponse é uma classe de uso flexível e transparente, mas se você precisar alterá-la por uma classe específica da aplicação, você poderá sobrescrevê-la e e substituí-la por sua própria classe, alterando o CakeResponse usado no arquivo index.php.

Isto fará com que todos os controllers da sua aplicação use CustomResponse ao invés de *CakeResponse*. Você pode também substituir a instancia utilizada, definindo o novo objeto em $this->response nos seus controllers. sobrescrever o objeto de resposta é útil durante os testes, permitindo você simular os métodos que interagem com o header(). Veja a seção *CakeResponse e Testes* para mais informações.

Lidando com tipos de conteúdo

Você pode controlar o «*Content-Type*» da resposta de sua aplicação usando o método *CakeResponse::type()*. Se sua aplicação precisa lidar com tipos de conteúdos que não estão inclusos no CakeResponse, você também poderá mapear estes tipos utilizando o método type():

```
// Adiciona o tipo vCard
$this->response->type(array('vcf' => 'text/v-card'));

// Define o Content-Type para vcard.
$this->response->type('vcf');
```

Normalmente você vai querer mapear os tipos de conteúdo adicionais no callback `beforeFilter` do seu controller, assim, se você estiver usando o `RequestHandlerComponent`, poderá tirar proveito da funcionalidade de troca de views baseado no tipo do conteúdo.

Enviando Anexos

Poderá existir momentos em que você queira enviar respostas dos controllers como sendo arquivos para downloads. Você pode conseguir este resultado usando *Media Views* ou usando as funcionalidades do `CakeResponse`. O método *`CakeResponse::download()`* permite você enviar respostas como arquivos para download:

```
function sendFile($id) {
    $this->autoRender = false;

    $file = $this->Attachment->getFile($id);
    $this->response->type($file['type']);
    $this->response->download($file['name']);
    $this->response->body($file['content']);
    $this->response->send();
}
```

O exemplo acima demonstra como você pode utilizar o `CakeResponse` para gerar um arquivo para download sem precisar usar a classe *`MediaView`*. Em geral, você vai preferir utilizar a classe `MediaView` por possuir maiores funcionalidades que o `CakeResponse`.

Interagindo com o cache do navegador

Algumas vezes você precisará forçar o browser do cliente a não fazer cache dos resultados de uma ação de um controller. *`CakeResponse::disableCache()`* é destinado para estes casos.:

```
function index() {
    // faz alguma coisa.
    $this->response->disableCache();
}
```

Aviso: Usar o `disableCache()` para downloads em domínios SSL enquanto tenta enviar arquivos para o Internet Explorer poderá resultar em erros.

Você também poderá dizer ao cliente para fazer cache da resposta. Usando *`CakeResponse::cache()`*:

```
function index() {
    // faz alguma coisa.
    $this->response->cache(time(), '+5 days');
}
```

O código acima diz aos clientes para armazenar em cache a resposta resultante por cinco dias, podendo acelerar a experiência dos seus visitantes.

Definindo Cabeçalhos

É possível definir cabeçalhos para a resposta utilizando o método *`CakeResponse::header()`*. Podendo ser chamada de algumas formas diferentes:

```
// Define um único cabeçalho
$this->response->header('Location', 'http://example.com');

// Define múltiplos cabeçalhos
$this->response->header(array('Location' => 'http://example.com', 'X-Extra' => 'My␣
↪header'));
$this->response->header(array('WWW-Authenticate: Negotiate', 'Content-type:␣
↪application/pdf'));
```

Definir o mesmo cabeçalho múltiplas vezes irá causar a sobrescrita do valor anterior, como numa chamada comum ao método `header()` do PHP. Os cabeçalhos não serão enviados quando o método *`CakeResponse::header()`* for chamado. Os cabeçalhos são armazenados em buffer até que a resposta seja efetivamente enviada.

CakeResponse e Testes

Provavelmente uma das grandes vitórias da classe `CakeResponse` vem de como ela torna mais fácil os testes de controllers e componentes. Ao invés de métodos espalhados em diversos objetos, você precisa de apenas um simples objeto para «forjar» e utilizar nos controllers e componentes. Isto lhe ajuda a criar seus testes unitários mais rapidamente:

```
function testSomething() {
    $this->controller->response = $this->getMock('CakeResponse');
    $this->controller->response->expects($this->once())->method('header');
    ...
}
```

Adicionalmente, você consegue testar sua aplicação pela linha de comando mais facilmente pois consegue «forjar» os cabeçalhos que quiser sem precisar ficar tentando definir os cabeçalhos diretos na interface de linha de comandos.

API do CakeResponse

`class CakeResponse`
A classe `CakeResponse` fornece vários métodos úteis para interagir com as respostas que você envia para um cliente.

`CakeResponse::`**`header`**`()`
Permite você definir diretamente um ou muitos cabeçalhos para serem enviados com a resposta.

`CakeResponse::`**`charset`**`()`
Define o mapa de caracteres (*charset*) que será usado na resposta.

`CakeResponse::`**`type`**`(`*$type*`)`
Define o tipo de conteúdo para a resposta. Você pode usar um apelido de um tipo conhecido de conteúdo ou usar um nome completo para o tipo do conteúdo.

`CakeResponse::`**`cache`**`()`
Permite você definir os cabeçalhos de cache em sua resposta.

`CakeResponse::`**`disableCache`**`()`
Define os cabeçalhos apropriados para desabilitar o cache da resposta pelo cliente.

`CakeResponse::`**`compress`**`()`
Habilita a compressão gzip para o envio da resposta.

`CakeResponse::`**`download`**`()`
Permite você enviar a resposta como um anexo e definir o nome do arquivo.

`CakeResponse::statusCode()`
: Permite você alterar o código do status da resposta.

`CakeResponse::body()`
: Define o conteúdo do corpo da resposta que será enviada.

`CakeResponse::send()`
: Após ter criado a resposta, chamar o método `send()` irá enviar os todos cabeçalhos definidos assim como o corpo da resposta. Isto é feito automaticamente no final de cada requisição pelo `Dispatcher`.

Fine tuning HTTP cache

Scaffolding (arcabouços)

O recurso de *scaffold* de aplicações é uma técnica que permite ao desenvolvedor definir e criar uma aplicação básica que possa inserir, selecionar, atualizar e excluir objetos. *scaffold* no CakePHP também possibilita que os desenvolvedores definam como os objetos estão relacionados entre si além de como criar e destruir estas relações.

Tudo o que é necessário para criar um *scaffold* é um model e seu controller correspondente. Uma vez que você tiver definido o atributo `$scaffold` em seu controller, este estará pronto para funcionar. O *scaffold* do CakePHP é muito legal. Ele permite que você tenha uma aplicação CRUD com tudo funcionando em minutos. É tão legal que você pode querer até usá-lo em produção. Podemos até achar isto legal também, mas por favor tenha em mente que *scaffold* é... ahn... apenas um arcabouço. O uso de *scaffold* poupa o trabalho da criação da estrutura real para acelerar o início de um projeto em etapas iniciais. *Scaffold* não tem intenção de ser completamente flexível, mas sim um jeito temporário de fazer as coisas funcionarem com brevidade. Se você se vir numa situação de querer personalizar a lógica e suas views, é hora de deixar de usar o recurso de *scaffold* e escrever o código de fato. A ferramenta de linha de comando Bake do CakePHP, abordado na próxima seção é um grande passo à frente: Ele gera todo o código que você deve precisar para produzir o mesmo resultado que teria atualmente com o scaffold.

Scaffold é uma excelente maneira de iniciar o desenvolvimento de partes prematuras da sua aplicação web. Primeiras versões de esquemas de bases de dados tendem a sofrer mudanças, o que é algo perfeitamente normal nas etapas iniciais do projeto da aplicação. Isto tem um lado negativo: Um desenvolvedor web detesta criar formulários que nunca virão a ser efetivamente usados. Para minimizar o esforço do desenvolvedor, o recurso de *scaffold* foi incluído no CakePHP. O *scaffold* analisa as tabelas de sua base de dados e cria uma listagem padronizada com botões de inserção, edição e exclusão, formulários padronizados para edição e views padronizadas para visualização de um único registro da base de dados.

Para adicionar o recurso de *scaffold* à sua aplicação, no controller, adicione o atributo `$scaffold`:

```
class CategoriesController extends AppController {
    public $scaffold;
}
```

Assumindo que você tenha criado um arquivo com a classe model mais básica para o `Category` (em `/app/Model/Category.php`), as coisas já estarão prontas. Acesse http://example.com/categories para ver sua nova aplicação com scaffold.

Nota: Criar métodos em controllers que possuam definições de *scaffold* pode causar resultados indesejados. Por exemplo, se você criar um método `index()` em um controller com scaffold, seu método index será renderizado no lugar da funcionalidade do scaffold.

O *scaffold* tem conhecimento sobre as associações de models, então se seu model `Category` possuir uma referência a (*belongsTo*) `User`, você verá os IDs dos usuários relacionados na listagem de `Category`. Enquanto o *scaffold* «sabe» como tratar os relacionamentos entre models, você não verá nenhum registro relacionado nas views do *scaffold* até que você tenha adicionado manualmente as relações entre os models. Por exemplo, se `Group` *hasMany* (possui

muitos) `User` e `User` *belongsTo* (pertence à) `Group`, você precisa adicionar manualmente o código necessário a seguir nos seus models `User` e `Group`. Antes de você adicionar o código a seguir, a view mostrará uma tag select vazia para `Group` no formulário de adição do model `User`. Após você adicionar o código, a view irá mostrar uma tag select populada com os IDs ou nomes vindos da tabela groups no formulário de adição de `User`.

```
// Em Group.php
public $hasMany = 'User';

// Em User.php
public $belongsTo = 'Group';
```

Se você preferir ver algo além do ID (como o primeiro nome dos usuários), você pode alterar o valor do atributo `$displayField` no model. Vamos ver o `$displayField` na nossa classe `User` de forma que os usuários relacionados com categorias sejam mostrados pelo primeiro nome ao invés de apenas o ID. Em muitos casos, este recurso torna o *scaffold* mais legível.

```
class User extends AppModel {
    public $name = 'User';
    public $displayField = 'first_name';
}
```

Criando uma interface administrativa simples com scaffolding

Se você tiver habilitado as rotas de *admin* em seu arquivo de configuração `app/Config/core.php` com a alteração a seguir `Configure::write('Routing.prefixes', array('admin'));`, você poderá usar o *scaffold* para gerar interfaces administrativas.

Uma vez que você tenha ativado a rota de *admin* atribua seu prefixo *admin* à variável scaffolding:

```
public $scaffold = 'admin';
```

Agora você poderá acessar o arcabouço de suas ações administrativas:

```
http://example.com/admin/controller/index
http://example.com/admin/controller/view
http://example.com/admin/controller/edit
http://example.com/admin/controller/add
http://example.com/admin/controller/delete
```

Esta é uma forma fácil de criar uma interface de administração simples rapidamente. Tenha em mente que você não pode ter ambos os métodos de scaffold, um para admin e outro para não-admin ao mesmo tempo. Assim como em um *scaffold* normal você pode sobrescrever um método individual com seu próprio código:

```
function admin_view($id = null) {
  // código customizado aqui
}
```

Uma vez que você tenha substituído uma ação de *scaffold* você também precisará criar um arquivo de view para a ação.

Customizando as Views de Scaffold

Se você quiser uma view de *scaffold* um pouco diferente, você pode criar templates. Continuamos a não recomendar o uso desta técnica para aplicações em produção, mas tal customização pode ser útil durante o período de prototipação.

A customização é feita criando templates de view:

Views de *scaffold* customizadas para um controller específico (PostsController neste example) devem ser colocadas no diretório das views desta maneira:

```
/app/View/Posts/scaffold.index.ctp
/app/View/Posts/scaffold.show.ctp
/app/View/Posts/scaffold.edit.ctp
/app/View/Posts/scaffold.new.ctp
```

Views de *scaffold* customizadas para todos os controllers devem ser criadas desta maneira:

```
/app/View/Scaffolds/index.ctp
/app/View/Scaffolds/form.ctp
/app/View/Scaffolds/view.ctp
```

O Controller Pages

O CakePHP já vem com um controller padrão chamado PagesController (`lib/Cake/Controller/PagesController.php`). A página inicial que você vê logo após a instalação é gerada usando este controller. Este controller é geralmente usado para servir páginas estáticas. Ex. Se você fez uma view `app/View/Pages/sobre_nos.ctp`, você pode acessá-la usando a seguinte URL http://example.com/pages/sobre_nos

Quando você constrói uma aplicação utilizando o console «bake» o controller Pages é copiado para seu diretório app/Controller/ e você pode modificá-lo se for preciso. Ou você pode fazer uma cópia do arquivo PagesController.php da pasta lib/Cake para seu diretório app/Controller/ existente.

> **Aviso:** Não modifique nenhum arquivo dentro do diretório `Cake` diretamente para evitar problemas futuros quando for atualizar o núcleo do framework CakePHP.

Componentes

Componentes (*components*) são pacotes com funções lógicas que são usadas para serem compartilhadas entre os controllers. Se você está querendo copiar e colar coisas entre seus controllers, talvez seja uma boa ideia considerar a possibilidade de empacotar estas funcionalidades em componentes.

O CakePHP também já vem com uma quantidade fantástica de componentes incluídos, que você pode usar para lhe ajudar com:

- Segurança
- Sessões
- Lista de Controle de Acesso (do inglês ACL, *Access control lists*)
- Emails
- Cookies
- Autenticação
- Tratamento de Requisições

Cada um destes componentes do Cake são detalhados em seus próprios capítulos. Neste momento, nós lhe mostraremos como criar e usar seus próprios componentes. Criar componentes mantem o código dos controllers limpos e permitem a reutilização de códigos entre projetos.

Configurando Componentes

Muitos dos componentes incluídos no Cake requerem alguma configuração. Exemplos de componentes que requerem configuração são: /core-libraries/components/authentication, /core-libraries/components/cookie. As configurações para estes componentes, e outros em geral, são feitas no array `$components` ou no método `beforeFilter()` do seu controller:

```
class PostsController extends AppController {
    public $components = array(
        'Auth' => array(
            'authorize' => array('controller'),
            'loginAction' => array('controller' => 'users', 'action' => 'login')
        ),
        'Cookie' => array('name' => 'CookieMonster')
    );
```

O exemplo acima seria um exemplo de como configurar um componente usando o array `$components`. Todos os componentes incluídos no Cake permitem ser configurados desta forma. Além disso, você pode configurar componentes no método `beforeFilter()` de seus controllers. Isto é útil quando você precisa atribuir os resultados de uma função para uma propriedade do componente. O exemplo acima também pode ser expressado da seguinte maneira:

```
public function beforeFilter() {
    $this->Auth->authorize = array('controller');
    $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');

    $this->Cookie->name = 'CookieMonster';
}
```

É possível, no entanto, que um componente requeira que certa configuração seja feita antes do método `beforeFilter()` do controller ser executado. Para este fim, alguns componentes permitem que configurações sejam feitas no array `$components`:

```
public $components = array('DebugKit.Toolbar' => array('panels' => array('history',
↪'session')));
```

Consulte a documentação relevante para determinar quais opções de configuração cada componente oferece.

Usando Componentes

Após ter incluído alguns componentes em seu controller, usá-los é muito simples. Cada componente que você usa é exposto como uma propriedade em seu controller. Se você carregou o `SessionComponent` e o `CookieComponent` no seu controller, você pode acessá-los da seguinte maneira:

```
class PostsController extends AppController {
    public $components = array('Session', 'Cookie');

    public function delete() {
        if ($this->Post->delete($this->request->data('Post.id')) {
            $this->Session->setFlash('Post deleted.');
            $this->redirect(array('action' => 'index'));
        }
    }
```

Nota: Como os models e componentes são adicionados no controller como propriedades eles compartilham o mesmo

espaço de nomes (`namespace`). Tenha certeza de não ter um componente e um model com o mesmo nome.

Carregando componentes sob demanda

Você pode não precisar de todos os componentes disponibilizados em cada ação dos controllers. Nestas situações você pode carregar um componente em tempo de execução usando o Component Collection. Dentro de um controller você pode fazer o seguinte:

```
$this->OneTimer = $this->Components->load('OneTimer');
$this->OneTimer->getTime();
```

Callbacks de Componentes

Componentes também oferecem alguns callbacks do ciclo de vida de uma requisição, permitindo acrescentar rotinas ao fluxo. Veja a *API dos Componentes* para mais informações sobre os callbacks que os componentes oferecem.

Criando um Componente

Suponhamos que nossa aplicação online precisa realizar uma operação matemática complexa em diferentes partes da aplicação. Podemos criar um componente para abrigar esta lógica para ser usada nos diferentes controllers.

O primeiro passo é criar um novo arquivo para a classe do componente. Crie o arquivo em `/app/Controller/Component/MathComponent.php`. A estrutura básica para o componente irá se parecer com algo assim:

```
class MathComponent extends Component {
    function doComplexOperation($amount1, $amount2) {
        return $amount1 + $amount2;
    }
}
```

Nota: Todos os componentes devem estender a classe *`Component`*. Se não fizer isto, o Cake irá disparar uma exceção.

Incluindo seus componentes nos seus controllers

Após nosso componente estiver pronto, podemos usá-lo nos controllers da nossa aplicação pondo o nome do componente (sem o sufixo «Component») no array `$components` do controller. O controller irá receber um novo atributo com o mesmo nome do componente, o qual poderemos acessá-lo como sendo uma instância da classe componente que queremos.

```
/* Torna o novo componente acessível em $this->Math,
bem como o $this->Session */
public $components = array('Math', 'Session');
```

Componentes declarados no `AppController` serão mesclados com os de outros controllers. Então não há necessidade de redeclarar o mesmo componente duas vezes.

Ao incluir componentes em um controller você também pode declarar um conjunto de parâmetros que serão passados para o construtor do componente. Estes parâmetros podem ser usados pelo componente.

```
public $components = array(
    'Math' => array(
        'precision' => 2,
        'randomGenerator' => 'srand'
    ),
    'Session', 'Auth'
);
```

O exemplo acima irá passar no segundo parâmetro do construtor `MathComponent::__construct()` um array contendo o atributo «precision» e «randomGenerator».

Por convenção, qualquer configuração que você tenha passado e que também seja um atributo público do seu componente, irá ter seu valor definido com base no array.

Usando outros componentes nos seus Componentes

Às vezes, um de seus componentes poderá precisar usar outro componente. Neste caso você pode incluir outros componentes no seu da mesma forma que inclui em controllers, usando o atributo `$components`:

```
// app/Controller/Component/CustomComponent.php
class CustomComponent extends Component {
    // O outro componente que seu componente utiliza
    public $components = array('Existing');

    function initialize(Controller $controller) {
        $this->Existing->foo();
    }

    function bar() {
        // ...
    }
}

// app/Controller/Component/ExistingComponent.php
class ExistingComponent extends Component {

    function initialize(Controller $controller) {
        $this->Parent->bar();
    }

    function foo() {
        // ...
    }
}
```

API dos Componentes

class Component

A classe base `Component` oferece alguns métodos para carregar sob demanda (Lazy loading. Possibilita adiar a inicialização de um objeto até que este seja utilizado) outros componentes utilizando o `ComponentCollection` assim como lidar com as configurações básicas. Esta classe também fornece os protótipos para todos os callbacks dos componentes.

`Component::__construct` (*ComponentCollection $collection*, *$settings = array()*)

O contrutor da classe `Component`. Todos as propriedades públicas da classe terão seus valores alterados para

corresponder com o valor de `$settings`.

Callbacks

`Component::initialize`(*$controller*)
O método `initialize` é chamado antes do método `beforeFilter` do controller.

`Component::startup`(*$controller*)
O método `startup` é chamado após o método `beforeFilter` do controller mas antes que o controller execute a ação.

`Component::beforeRender`(*$controller*)
O método `beforeRender` é chamado após o controller executar a lógica da ação requisitada mas antes que o controller renderize a view e o layout.

`Component::shutdown`(*$controller*)
O método `shutdown` é chamado antes que o conteúdo seja enviado para o browser.

`Component::beforeRedirect`(*$controller*, *$url*, *$status=null*, *$exit=true*)
O método `beforeRedirect` é invocado quando o método `redirect` de um controller é chamado mas antes de qualquer ação. Se este método retornar `false` o controller não irá continuar com o redirecionamento. As variáveis `$url`, `$status` e `$exit` possuem o mesmo significado do método do controller. Você pode também retornar uma string que será interpretada como uma URL para ser usada no redirecionamento ou retornar um array associativo com a chave "url" e opcionalmente com a chave "status" e a chave "exit".

CAPÍTULO 5

Views

Views (ou Visões) são o **V** do MVC. *Views* são responsáveis por gerar a saída de dados específica para uma determinada requisição. Geralmente esta saída é apresentada na forma de HTML, XML ou JSON. No entanto, disponibilizar arquivos através de *streaming* (fluxo de informação, geralmente multimídia, através de pacotes) ou criar PDFs, que podem ser baixados, também são de responsabilidade da Camada *View*.

O CakePHP traz incluso várias classes do tipo *View* para lidar com os cenários mais comuns de renderização:

- Para criar *webservices* em XML ou JSON, você pode usar o *JSON and XML views*
- Para prover arquivos protegidos ou arquivos criados dinamicamente, você pode usar *Media Views*
- Para criar múltiplos temas para as visões, você pode usar *Themes*

View Templates

Em CakePHP, você fala com seus usuários através da camada *view* (visão). Na maior parte do tempo, suas *views* exibirão documentos (X)HTML nos navegadores, mas você pode também precisar prover dados AMF para um objeto em Flash, responder a uma aplicação remota via SOAP ou gerar um arquivo CSV para um usuário.

Por padrão, no CakePHP os arquivos do tipo *view* são escritos em PHP comum e possuem a extensão .ctp (CakePHP *Template*). Estes arquivos contém toda a lógica de apresentação necessária para transformar os dados recebidos do *controller* em um formato pronto para o público. Caso você prefira usar uma linguagem de *template* como Twig ou Smarty, uma subclasse da *View* irá fazer uma ponte entre sua linguagem de *template* e o CakePHP.

Arquivos do tipo *view* são guardados em `/app/View/`, dentro do diretório com o nome do *controller* que usa os arquivos e nomeado de acordo com a ação correspondente. Por exemplo, a ação "view()" do *controller Products* será normalmente encontrada em `/app/View/Products/view.ctp`.

A camada *view* no CakePHP pode ser composta de diferentes partes. Cada parte tem diferentes usos e serão cobertas em seções específicas:

- **views**: *views* é a única parte da página que está em execução. Compõem a parte crucial da resposta da aplicação.

- **elements**: pedaços de código pequenos e reutilizáveis. *Elements* geralmente são renderizados dentro de *views*.
- **layouts**: arquivos da *view* contendo código de apresentação que envolve várias interfaces da aplicação. A maior parte dos arquivos *views* é renderizada dentro de um *layout*.
- **helpers**: essas classes encapsulam lógica da *view* que seja necessária em vários lugares na camada *view*. *Helpers* no CakePHP podem ajudá-lo a construir formulários, construir funcionalidade AJAX, paginar dados do *model*, prover *feeds* RSS, dentre outras coisas.

Estendendo Views

Novo na versão 2.1.

A extensão de uma *View* permite que você inclua uma *view* dentro de outra. Combinando isto com *view blocks* você tem uma maneira poderosa para deixar suas *views DRY* (enxutas). Por exemplo, sua aplicação tem uma barra lateral (*sidebar*) que precisa mudar a depender de quando uma *view* específica é renderizada. Estendendo um mesmo arquivo de *view*, você pode evitar repetições de marcações em comum e apenas definir as que mudam:

```
// app/View/Common/view.ctp
<h1><?php echo $this->fetch('title'); ?></h1>
<?php echo $this->fetch('content'); ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
    <?php echo $this->fetch('sidebar'); ?>
    </ul>
</div>
```

O arquivo de *view* acima pode ser usado como uma *view* pai. Esta espera que a *view* que a estende defina os blocos `sidebar` e `title`. O bloco `content` é um bloco especial que o CakePHP cria. Ele conterá todo o conteúdo não-capturado da *view* que a estende. Considerando que nosso arquivo *view* tem uma variável `$post` com informação sobre nosso *post*, nossa *view* poderá parecer como:

```
<?php
// app/View/Posts/view.ctp
$this->extend('/Common/view');

$this->assign('title', $post)

$this->start('sidebar');
?>
<li>    echo $this->Html->link('edit', array(
    'action' => 'edit',
    $post['Post']['id']
)); ?>
</li>
<?php $this->end(); ?>


// O conteúdo restante estará disponível como o bloco `content`
// na view pai.
echo h($post['Post']['body']);
```

A *view* de *post* acima mostra como você pode estender uma *view* e preenche-la com um conjunto de blocos. Qualquer conteúdo que não estiver definido em um bloco será capturado e colocado em um bloco especial chamado `content`. Quando uma *view* contém uma chamada para `extend()`, a execução continua até o fim do arquivo *view* atual. Uma

vez finalizada, a *view* estendida será renderizada. Chamar `extend()` mais de uma vez em um arquivo *view* irá sobrescrever a *view* pai que será processada em seguida:

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

O trecho acima resultará em `/Common/index.ctp` sendo renderizada como a *view* pai para a *view* atual.

Você pode aninhar *views* estendidas quantas vezes forem necessárias. Cada *view* pode estender outra *view* se quiser. Cada *view* pai pegará o conteúdo da *view* anterior como o bloco `content`.

Nota: Você deve evitar o uso de `content` como o nome de um bloco em sua aplicação. CakePHP usa este nome em *views* estendidas para conteúdos não-capturados.

Usando Blocos de Views (Visões)

Novo na versão 2.1.

Blocos de *views* substituem `$scripts_for_layout` e provêm uma API flexível que permite criar *slots* ou blocos em suas *views/layouts* que podem ser definidas em qualquer lugar. Por exemplo, blocos são ideais para implementar recursos como barras laterais ou regiões para carregar seções na parte de baixo ou no topo do *layout*. Blocos podem ser definidos de duas formas. Seja capturando um bloco ou por atribuição direta. Os métodos `start()`, `append()` e `end()` permitem trabalhar com captura de blocos:

```
// cria um bloco lateral.
$this->start('sidebar');
echo $this->element('sidebar/recent_topics');
echo $this->element('sidebar/recent_comments');
$this->end();


// Concatena na barra lateral em seguida.
$this->append('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();
```

Também é possível concatenar blocos utilizando o método `start()` múltiplas vezes. O método `assign()` pode ser usado para limpar ou sobrescrever o bloco:

```
// Limpa o conteúdo anterior da barra lateral.
$this->assign('sidebar', '');
```

Nota: Você deve evitar o uso de `content` como o nome de um bloco em sua aplicação. CakePHP usa este nome em *views* estendidas para conteúdos não-capturados .

Exibindo blocos

Novo na versão 2.1.

Você pode exibir blocos usando o método `fetch()`. `fetch()` irá retornar um bloco de maneira segura, retornando "" se o bloco não existir»:

```
echo $this->fetch('sidebar');
```

Você também pode usar o *fetch* para exibir condicionalmente um conteúdo que deve envolver um bloco que deveria existir. Isto é útil em *layouts* ou *views* estendidas, nas quais você queira mostrar cabeçalhos e outras marcações condicionalmente:

```
// em app/View/Layouts/default.ctp
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?php echo $this->fetch('menu'); ?>
</div>
<?php endif; ?>
```

Utilizando blocos para arquivos de script e CSS

Novo na versão 2.1.

Blocos substituem a variável obsoleta `$scripts_for_layout` do *layout*. Em vez de usá-la, você deve usar blocos. A `HtmlHelper` vincula-se aos blocos da *view* e a cada um dos seus métodos php:meth:*~HtmlHelper::script()*, `css()` e `meta()` quando o bloco com o mesmo nome utiliza a opção `inline = false`:

```
<?php
// no seu arquivo de view
$this->Html->script('carousel', array('inline' => false));
$this->Html->css('carousel', array('inline' => false));
?>

// no seu arquivo de layout
<!DOCTYPE html>
<html lang="en">
    <head>
    <title><?php echo $this->fetch('title'); ?></title>
    <?php echo $this->fetch('script'); ?>
    <?php echo $this->fetch('css'); ?>
    </head>

    // o resto do layout continua
```

A `HtmlHelper` também permite você controlar para que bloco os *scripts* e CSS vão:

```
// na sua view
$this->Html->script('carousel', array('block' => 'scriptBottom'));

// no seu layout
echo $this->fetch('scriptBottom');
```

Layouts

Um *layout* contem o código de apresentação que envolve uma *view*. Qualquer coisa que você queira ver em todas as suas *views* deve ser colocada em um *layout*.

Arquivos de *layouts* devem ser colocados em `/app/View/Layouts`. O *layout* padrão do CakePHP pode ser sobrescrito criando um novo *layout* padrão em `/app/View/Layouts/default.ctp`. Uma vez que um novo *layout*

padrão tenha sido criado, o código da *view* renderizado pelo *controller* é colocado dentro do *layout* padrão quando a página é renderizada.

Quando você cria um *layout*, você precisa dizer ao CakePHP onde colocar o código de suas *views*. Para isso, garanta que o seu *layout* inclui um lugar para `$this->fetch('content')`. A seguir, um exemplo de como um *layout* padrão deve parecer:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?php echo $title_for_layout?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Incluir arquivos extenos e scripts aqui (Ver o helper HTML para mais detalhes) --
↪>
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
?>
</head>
<body>

<!-- Se você quiser exibir algum menu
em todas as suas views, inclua-o aqui -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Aqui é onde eu quero que minhas views sejam exibidas -->
<?php echo $this->fetch('content'); ?>

<!-- Adicionar um rodapé para cada página exibida -->
<div id="footer">...</div>

</body>
</html>
```

Nota: Na versão anterior a 2.1, o método fetch() não estava disponível, `fetch('content')` é uma substituição para `$content_for_layout` e as linhas `fetch('meta')`, `fetch('css')` and `fetch('script')` estavam contidas na variável `$scripts_for_layout` na versão 2.0.

Os blocos `script`, `css` e `meta` contém qualquer conteúdo definido nas *views* usando o *helper* HTML embutido. Útil na inclusão de arquivos *javascript* e CSS de *views*.

Nota: Quando usar `HtmlHelper::css()` ou `HtmlHelper::script()` em *views*, especifique "false" para a opção "inline" para colocar o código html em um bloco de mesmo nome. (Veja a API para mais detalhes de uso)

O bloco `content` contem o conteúdo da *view* renderizada.

`$title_for_layout` contém o título da página, Esta variável é gerada automaticamente, mas você poderá sobrescrevê-la definindo-a em seu *controller*/*view*.

Para definir o título para o *layout*, o modo mais fácil é no *controller*, setando a variável `$title_for_layout`:

```
class UsersController extends AppController {
    public function view_active() {
        $this->set('title_for_layout', 'View Active Users');
```

```
    }
}
```

Você também pode setar a variável title_for_layout no arquivo de *view*:

```
$this->set('title_for_layout', $titleContent);
```

Você pode criar quantos *layouts* você desejar: apenas coloque-os no diretório `app/View/Layouts`, e defina qual deles usar dentro das ações do seu *controller* usando a propriedade *`$layout`* do *controller* ou *view*:

```
// de um controller
public function admin_view() {
    // códigos
    $this->layout = 'admin';
}

// de um arquivo view
$this->layout = 'loggedin';
```

Por exemplo, se a seção do meu *site* incluir um pequeno espaço para *banner*, eu posso criar um novo *layout* com um pequeno espaço para propaganda e especificá-lo como *layout* para as ações de todos os *controllers* usando algo como:

```
class UsersController extends AppController {
    public function view_active() {
        $this->set('title_for_layout', 'View Active Users');
        $this->layout = 'default_small_ad';
    }

    public function view_image() {
        $this->layout = 'image';
        //output user image
    }
}
```

O CakePHP tem em seu núcleo, dois *layouts* (além do *layout* padrão) que você pode usar em suas próprias aplicações: "ajax" e "flash". O *layout* Ajax é útil para elaborar respostas Ajax - é um *layout* vazio (a maior parte das chamadas ajax requer pouca marcação de retorno, preferencialmente a uma interface totalmente renderizada). O *layout* flash é usado para mensagens mostradas pelo método *`Controller::flash()`*.

Outros três *layouts*, XML, JS, e RSS, existem no núcleo como um modo rápido e fácil de servir conteúdo que não seja text/html.

Usando layouts a partir de plugins

Novo na versão 2.1.

Se você quiser usar um *layout* que existe em um *plugin*, você pode usar a sintaxe de *plugin*. Por exemplo, para usar o *layout* de contato do *plugin* de contatos:

```
class UsersController extends AppController {
    public function view_active() {
        $this->layout = 'Contacts.contact';
    }
}
```

Elements

Muitas aplicações possuem pequenos blocos de código de apresentação que precisam ser repetidos a cada página, às vezes em diferentes lugares no *layout*. O CakePHP ajuda você a repetir partes do seu *website* que precisam ser reutilizados. Estas partes reutilizáveis são chamadas de *Elements* (ou Elementos). Propagandas, caixas de ajuda, controles de navegação, *menus* extras, formulários de *login* e chamadas geralmente são implementadas como *elements*. Um *element* é básicamente uma *mini-view* que pode ser incluída em outras *views*, *layouts* e até mesmo em outros *elements*. *Elements* podem ser usados para criar uma *view* mais legível, colocando o processamento de elementos repetidos em seu próprio arquivo. Eles também podem ajudá-lo a re-usar conteúdos fragmentados pela sua aplicação.

Elements são colocados na pasta `/app/View/Elements/` e possuem a extensão .ctp no nome do arquivo. Eles são exibidos através do uso do método *element* da *view*:

```
echo $this->element('helpbox');
```

Passando variáveis em um Element

Você pode passar dados para um *element* através do segundo argumento do *element*:

```
echo $this->element('helpbox', array(
    "helptext" => "Oh, este texto é muito útil."
));
```

Dentro do arquivo do *element*, todas as variáveis passadas estão disponíveis como membros do *array* de parâmetros (da mesma forma que *Controller::set()* no *controller* trabalha com arquivos de *views*). No exemplo acima, o arquivo `/app/View/Elements/helpbox.ctp` pode usar a variável `$helptext`:

```
// Dentro de app/View/Elements/helpbox.ctp
echo $helptext; //outputs "Oh, este texto é muito útil."
```

O método *View::element()* também suporta opções para o *element*. As opções suportadas são "cache" e "callbacks". Um exemplo:

```
echo $this->element('helpbox', array(
        "helptext" => "Isto é passado para o *element * como $helptext",
        "foobar" => "TIsto é passado para o *element * como $foobar",
    ),
    array(
        "cache" => "long_view", // usa a configuração de cache "long_view"
        "callbacks" => true // atribue verdadeiro para ter before/afterRender chamado␣
↪pelo *element*
    )
);
```

O *cache* de *element* é facilitado através da classe `Cache`. Você pode configurar *elements* para serem guardados em qualquer configuração de *cache* que você tenha definido. Isto permite uma maior flexibilidade para decidir onde e por quantos *elements* são guardados. Para fazer o *cache* de diferentes versões de um mesmo *element* em uma aplicação, defina uma única chave de *cache* usando o seguinte formato:

```
$this->element('helpbox', array(), array(
        "cache" => array('config' => 'short', 'key' => 'unique value')
    )
);
```

Você pode tirar vantagem de *elements* usando `requestAction()`. A função `requestAction()` carrega variáveis da *views* a partir de ações do *controller* e as retorna como um *array*. Isto habilita seus *elements* para atuar verdadeiramente no estilo MVC. Crie uma ação de *controller* que prepara as variáveis da *view* para seu *element*, depois chame `requestAction()` no segundo parâmetro do `element()` para carregar as variáveis da *view* a partir do seu *controller*.

Para isto, em seu *controller*, adicione algo como segue, como exemplo de *Post*:

```
class PostsController extends AppController {
    // ...
    public function index() {
        $posts = $this->paginate();
        if ($this->request->is('requested')) {
            return $posts;
        } else {
            $this->set('posts', $posts);
        }
    }
}
```

Em seguida, no *element*, você poderá acessar os modelos de *posts* paginados. Para obter os últimos cinco *posts* em uma lista ordenada, você pode fazer algo como:

```
<h2>Latest Posts</h2>
<?php $posts = $this->requestAction('posts/index/sort:created/direction:asc/limit:5');
↪ ?>
<?php foreach ($posts as $post): ?>
<ol>
    <li><?php echo $post['Post']['title']; ?></li>
</ol>
<?php endforeach; ?>
```

Caching Elements

Você pode tomar proveito do CakePHP *view caching*, se você fornecer um parâmetro de *cache*. Se definido como *true*, o *element* será guardado na configuração de *cache* "default". Caso contrário, você poderá definir qual configuração de *cache* deve ser usada. Veja /core-libraries/caching para mais informações de configuração `Cache`. Um exemplo simples de *caching* um *element* seria:

```
echo $this->element('helpbox', array(), array('cache' => true));
```

Se você renderiza o mesmo *element* mais que uma vez em uma *view* e tem *caching* ativado, esteja certo de definir o parâmetro chave (*key*) para um nome diferente cada vez. Isto irá prevenir que cada chamada sucessiva substitua o resultado armazenado da chamada element() anterior. E.g.:

```
echo $this->element(
    'helpbox',
    array('var' => $var),
    array('cache' => array('key' => 'first_use', 'config' => 'view_long')
);

echo $this->element(
    'helpbox',
    array('var' => $differenVar),
    array('cache' => array('key' => 'second_use', 'config' => 'view_long')
);
```

O código acima garante que ambos os resultados do *element* serão armazenados separadamente. Se você quiser que todos os elementos armazenados usem a mesma configuração de *cache*, você pode salvar alguma repetição, setando `View::$elementCache` para a configuração de *cache* que você quer usar. O CakePHP usará esta configuração, quando nenhuma outra for dada.

Requisitando Elements de um Plugin

2.0

Para carregar um *element* de um *plugin*, use a opção *plugin* (retirada da opção *data* na versão 1.x):

```
echo $this->element('helpbox', array(), array('plugin' => 'Contacts'));
```

2.1

Se você está usando um *plugin* e deseja usar *elements* de dentro deste *plugin* apenas use *plugin syntax*. Se a *view* está renderizando para um *controller/action* de *plugin*, o nome do *plugin* será automaticamente prefixado antes de todos os *elements* usados, ao menos que outro nome de *plugin* esteja presente. Se o *element* não existir no *plugin*, será procurado na pasta principal da APP.:

```
echo $this->element('Contacts.helpbox');
```

Se sua *view* é parte de um *plugin* você pode omitir o nome do *plugin*. Por exemplo, se você está no `ContactsController` do *plugin* Contatos:

```
echo $this->element('helpbox');
// and
echo $this->element('Contacts.helpbox');
```

São equivalentes e resultarão no mesmo elemento sendo renderizado.

Alterado na versão 2.1: A opção `$options[plugin]` foi descontinuada e o suporte para `Plugin.element` foi adicionado.

View API

class View

Métodos de *Views* são acessíveis por todas as *views*, *elements* e arquivos de *layout*. Para chamar qualquer método de uma *view* use `$this->method()`.

`View::set`(*string $var*, *mixed $value*)
: *Views* têm métodos `set()` que são análogos aos `set()` encontrados nos objetos *controllers*. Usando set() em seu arquivo *view* serão adicionados variáveis para *layouts* e *elements* que serão renderizados posteriormente. Veja *Métodos dos Controllers* para maiores informações de como usar o set().

  No seu arquivo de *view*, você pode:

  ```
  $this->set('activeMenuButton', 'posts');
  ```

  Assim em seu *layout* a variável `$activeMenuButton` estará disponível e conterá o valor "posts".

`View::getVar`(*string $var*)
: Obtem o valor de *viewVar* com o nome $var

`View::getVars()`
Obtem uma lista de todas as variáveis disponíveis da *view*, no escopo renderizado corrente. Retorna um *array* com os nomes das variáveis.

`View::element(string $elementPath, array $data, array $options = array())`
Renderiza um elemento ou parte de uma *view*. Veja a seção *Elements* para maiores informações e exemplos.

`View::uuid(string $object, mixed $url)`
Gera um DOM ID não randômico único para um objeto, baseado no tipo do objeto e url. Este método é frequentemente usado por *helpers* que precisam gerar DOM ID únicos para elementos como `JsHelper`:

```
$uuid = $this->uuid('form', array('controller' => 'posts', 'action' => 'index'));
//$uuid contains 'form0425fe3bad'
```

`View::addScript(string $name, string $content)`
Adiciona conteúdo para *buffer* de *scripts* internos. Este *buffer* é disponibilizado no *layout* como `$scripts_for_layout`. Este método auxilia na criação de *helpers* que necessitam adicionar javascript or css diretamente para o *layout*. Ciente que *scripts* adicionados de *layouts*, or *elements* do *layout* não serão adicionados para `$scripts_for_layout`. Este método é frequentemente usado dentro dos *helpers*, como nos *Helpers /core-libraries/helpers/js* e /core-libraries/helpers/html.

Obsoleto desde a versão 2.1: Use a *feature* *Usando Blocos de Views (Visões)*, ao invés.

`View::blocks()`
Obtem o nome de todos os blocos definidos como um *array*.

`View::start($name)`
Inicia a caputura de bloco para um bloco de *view*. Veja a seção em *Usando Blocos de Views (Visões)* para exemplos.

Novo na versão 2.1.

`View::end()`
Finaliza o mais recente bloco sendo capturado. Veja a seção em *Usando Blocos de Views (Visões)* para exemplos.

Novo na versão 2.1.

`View::append($name, $content)`
Anexa no bloco com `$name`. Veja a seção em *Usando Blocos de Views (Visões)* para examplos.

Novo na versão 2.1.

`View::assign($name, $content)`
Atribui o valor de um bloco. Isso irá sobrescrever qualquer conteúdo existente. Veja a seção em *Usando Blocos de Views (Visões)* para exemplos.

Novo na versão 2.1.

`View::fetch($name)`
Fetch o valor do bloco. "" Serão retornados de blocos que não estão definidos Veja a seção em *Usando Blocos de Views (Visões)* para exemplos.

Novo na versão 2.1.

`View::extend($name)`
Estende o *view/element/layout* corrente com o nome fornecido. Veja a seção em *Estendendo Views* para examplos.

Novo na versão 2.1.

property `View::$layout`
Seta o *layout* onde a *view* corrente será envolvida.

property View::$**elementCache**
A configuração de *cache* usada para armazenar *elements*. Setando esta propriedade a configuração padrão usada para armazenar *elements* será alterada Este padrão pode ser sobrescrito usando a opção "cache" no método do *element*.

property View::$**request**
Uma instância de *CakeRequest*. Use esta instância para acessar informaçãoes sobre a requisição atual.

property View::$**output**
Contem o último conteúdo renderizado de uma *view*, seja um arquivo de *view* ou conteúdo do *layout*.

Obsoleto desde a versão 2.1: Use `$view->Blocks->get('content');` ao invés.

property View::$**Blocks**
Uma instância de `ViewBlock`. Usada para prover um bloco de funcionalidades de *view* na *view* renderizada.

Novo na versão 2.1.

More about Views

Themes

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[33] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Media Views

class MediaView

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[34] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

JSON and XML views

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[35] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[33] https://github.com/cakephp/docs
[34] https://github.com/cakephp/docs
[35] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Helpers

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[36] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

[36] https://github.com/cakephp/docs

CAPÍTULO 6

Plugins

CakePHP permite que você defina uma combinação de controllers, models, e views e lance-os como um pacote de aplicação que outras pessoas podem usar em suas aplicações CakePHP. Você quer ter um módulo de gestão de usuários, um blog simples, ou um módulo de serviços web em uma das suas aplicações? Empacote-os como um plugin do CakePHP para que você possa colocá-lo em outras aplicações.

A principal diferença entre um plugin e a aplicação em que ele é instalado, é a configuração da aplicação (base de dados, conexão, etc.). Caso contrário, ele opera em seu próprio espaço, comportando-se como se fosse uma aplicação por conta própria.

How To Install Plugins

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[37] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

How To Use Plugins

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

[37] https://github.com/cakephp/docs

Por favor, sinta-se a vontade para nos enviar um pull request no Github[38] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

How To Create Plugins

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[39] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Instalando um Plugin

Para instalar um plugin, basta simplesmente colocar a pasta plugin dentro do diretório app/Plugin. Se você está instalando um plugin chamado "ContactManager", então você deve ter uma pasta dentro de app/Plugin chamado "ContactManager" em que podem estar os diretórios de plugin: View, Model, Controller, webroot, e qualquer outro diretório.

Novo para CakePHP 2.0, plugins precisam ser carregados manualmente em app/Config/bootstrap.php.

Você pode carregá-los um por um ou todos eles em uma única chamada:

```
CakePlugin::loadAll(); // Carrega todos os plugins de uma vez
CakePlugin::load('ContactManager'); // Carrega um único plugin
```

loadAll carrega todos os plugins disponíveis, enquanto permite que você defina certas configurações para plugins específicos. load() funciona de maneira semelhante, mas carrega somente os plugins que você especificar explicitamente.

Há uma porção de coisas que você pode fazer com os métodos load e loadAll para ajudar com a configuração do plugin e roteamento. Talvez você tenha que carregar todos os plugins automaticamente, enquanto especifica rotas personalizadas e arquivos de bootstrap para certos plugins.

Sem problema:

```
CakePlugin::loadAll(array(
    'Blog' => array('routes' => true),
    'ContactManager' => array('bootstrap' => true),
    'WebmasterTools' => array('bootstrap' => true, 'routes' => true),
));
```

Com este estilo de configuração, você não precisa mais fazer manualmente um include() ou require() de arquivo de configuração do plugin ou rotas – acontece automaticamente no lugar e na hora certa. Os exatos mesmos parâmetros também poderiam ter sido fornecidos ao método load(), o que teria carregado apenas aqueles três plugins, e não o resto.

[38] https://github.com/cakephp/docs
[39] https://github.com/cakephp/docs

Finalmente, você pode especificar também um conjunto de padrões para loadAll que se aplicará a cada plugin que não tem uma configuração mais específica.

Carrega o arquivo bootstrap de todos os plugins, e as rotas do plugin Blog:

```
CakePlugin::loadAll(array(
    array('bootstrap' => true),
    'Blog' => array('routes' => true)
));
```

Note que todos os arquivos especificados devem realmente existir no plugin(s) configurado ou o PHP irá dar avisos (warnings) para cada arquivo que não pôde carregar. Isto é especialmente importante para lembrar ao especificar padrões para todos os plugins.

Alguns plugins precisam que seja criado uma ou mais tabelas em sua base de dados. Nesses casos, muitas vezes eles vão incluir um arquivo de esquema que você pode chamar a partir do cake shell dessa forma:

```
user@host$ cake schema create --plugin ContactManager
```

A maioria dos plugins indicará o procedimento adequado para a configurá-los e configurar a base de dados, em sua documentação. Alguns plugins irão exigir mais configurações do que outros.

Usando um Plugin

Você pode referenciar controllers, models, components, behaviors, e helpers de um plugin, prefixando o nome do plugin antes do nome da classe.

Por exemplo, digamos que você queira usar o ContactInfoHelper do plugin CantactManager para a saída de algumas informações de contato em uma de suas views. Em seu controller, seu array $helpers poderia ser assim:

```
public $helpers = array('ContactManager.ContactInfo');
```

Você então será capaz de acessar o ContactInfoHelper como qualquer outro helper em sua view, tal como:

```
echo $this->ContactInfo->address($contact);
```

Criando Seus Próprios Plugins

Como um exemplo de trabalho, vamos começar a criar o plugin ContactManager referenciado acima. Para começar, vamos montar a nossa estrutura básica de plugins. Ela deve se parecer com isso:

```
/app
    /Plugin
        /ContactManager
            /Controller
                /Component
            /Model
                /Behavior
            /View
                /Helper
                /Layouts
```

Note o nome da pasta do plugin, "**ContactManager**". É importante que essa pasta tenha o mesmo nome do plugin.

Dentro da pasta do plugin, você verá que se parece muito com uma aplicação CakePHP, e que basicamente é isso mesmo. Você realmente não tem que incluir qualquer uma dessas pastas se você não for usá-las. Alguns plugins podem definir somente um Component e um Behavior, e nesse caso eles podem omitir completamente o diretório "View".

Um plugin pode também ter basicamente qualquer um dos outros diretórios que sua aplicação pode, como Config, Console, Lib, webroot, etc.

Nota: Se você quer ser capaz de acessar seu plugin com uma URL, é necessário definir um AppController e AppModel para o plugin. Estas duas classes especiais são nomeadas após o plugin, e estendem AppController e AppModel da aplicação pai. Aqui está o que deve ser semelhante para nosso exemplo ContactManager:

```
// /app/Plugin/ContactManager/Controller/ContactManagerAppController.php:
class ContactManagerAppController extends AppController {
}
```

```
// /app/Plugin/ContactManager/Model/ContactManagerAppModel.php:
class ContactManagerAppModel extends AppModel {
}
```

Se você se esqueceu de definir estas classes especiais, o CakePHP irá entregar a você erros «Missing Controller» até que você tenha feito isso.

Por favor, note que o processo de criação de plugins pode ser muito simplificado usando o Cake shell.

Para assar um plugin por favor use o seguinte comando:

```
user@host$ cake bake plugin ContactManager
```

Agora você pode assar usando as mesmas convenções que se aplicam ao resto de sua aplicação. Por exemplo - assando controllers:

```
user@host$ cake bake controller Contacts --plugin ContactManager
```

Por favor consulte o capítulo /console-and-shells/code-generation-with-bake se você tiver quaisquer problemas com o uso da linha de comando.

Plugin Controllers

Controllers de nosso plugin ContactManager serão armazenados em /app/Plugin/ContactManager/Controller/. Como a principal coisa que vamos fazer é a gestão de contatos, vamos precisar de um ContactsController para este plugin.

Então, nós colocamos nosso novo ContactsController em /app/Plugin/ContactManager/Controller e deve se parecer com isso:

```
// app/Plugin/ContactManager/Controller/ContactsController.php
class ContactsController extends ContactManagerAppController {
    public $uses = array('ContactManager.Contact');

    public function index() {
        //...
    }
}
```

Nota: Este controller estende o AppController do plugin (chamado ContactManagerAppController) ao invés do AppController da aplicação pai.

Observe também como o nome do model é prefixado com o nome do plugin. Isto é necessário para diferenciar entre models do plugin e models da aplicação principal.

Neste caso, o array $uses não seria necessário com ContactManager. Contact seria o model padrão para este controller, no entanto está incluído para demostrar adequadamente como preceder o nome do plugin.

Se você quiser acessar o que nós fizemos até agora, visite /contact_manager/contacts. Você deve obter um erro «Missing Model» porque não temos um model Contact definido ainda.

Plugin Models

Models para plugins são armazenados em /app/Plugin/ContactManager/Model. Nós já definimos um ContactsController para este plugin, então vamos criar o model para o controller, chamado Contact:

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
}
```

Visitando /contact_manager/contacts agora (dado que você tem uma tabela em seu banco de dados chamada 'contacts') deveria nos dar um erro "Missing View". Vamos criar na próxima.

Nota: Se você precisar fazer referência a um model dentro de seu plugin, você precisa incluir o nome do plugin com o nome do model, separados por um ponto.

Por exemplo:

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
    public $hasMany = array('ContactManager.AltName');
}
```

Se você preferir que as chaves do array para associação não tenha o prefixo do plugin nelas, use uma sintaxe alternativa:

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
    public $hasMany = array(
        'AltName' => array(
            'className' => 'ContactManager.AltName'
        )
    );
}
```

Plugin Views

Views se comportam exatamente como fazem em aplicações normais. Basta colocá-las na pasta certa dentro de /app/Plugin/[PluginName]/View/. Para nosso plugin ContactManager, vamos precisar de uma view para nosso action ContactsController::index(), por isso vamos incluir isso como:

```
// /app/Plugin/ContactManager/View/Contacts/index.ctp:
<h1>Contacts</h1>
<p>Following is a sortable list of your contacts</p>
<!-- A sortable list of contacts would go here....-->
```

Nota: Para obter informações sobre como usar elements de um plugin, veja *Elements*

Substituindo views de plugins de dentro da sua aplicação

Você pode substituir algumas views de plugins de dentro da sua app usando caminhos especiais. Se você tem um plugin chamado "ContactManager" você pode substituir os arquivos de view do plugin com lógicas de view da aplicação específica criando arquivos usando o modelo a seguir «app/View/Plugin/[Plugin]/[Controller]/[view].ctp». Para o controller Contacts você pode fazer o seguinte arquivo:

```
/app/View/Plugin/ContactManager/Contacts/index.ctp
```

A criação desse, permite a você substituir «/app/Plugin/ContactManager/View/Contacts/index.ctp».

Imagens de Plugin, CSS e Javascript

Imagens, css e javascript de um plugin (mas não arquivos PHP), podem ser servidos por meio do diretório de plugin "webroot", assim como imagens, css e javascript da aplicação principal:

```
app/Plugin/ContactManager/webroot/
                                    css/
                                    js/
                                    img/
                                    flash/
                                    pdf/
```

Você pode colocar qualquer tipo de arquivo em qualquer diretório, assim como um webroot normal. A única restrição é que `MediaView` precisa saber o mime-type do arquivo.

Linkando para imagens, css e javascript em plugins

Basta preceder /plugin_name/ no início de um pedido para um arquivo dentro do plugin, e ele vai funcionar como se fosse um arquivo do webroot de sua aplicação.

Por exemplo, linkando para "/contact_manager/js/some_file.js" deveria servir o arquivo "app/Plugin/ContactManager/webroot/js/some_file.js".

Nota: É importante notar o **/your_plugin/** prefixado antes do caminho do arquivo. Isso faz a magica acontecer!

Components, Helpers e Behaviors

Um plugin pode ter Conponents, Helpers e Behaviors como uma aplicação CakePHP normal. Você pode até criar plugins que consistem apenas de Components, Helpers ou Behaviors que podem ser uma ótima maneira de contruir

componentes reutilizáveis que podem ser facilmente acoplados em qualquer projeto.

A construção destes componentes é exatamente o mesmo que contruir dentro de uma aplicação normal, sem convenção especial de nomenclatura.

Referindo-se ao seu componente de dentro ou fora do seu plugin, exige somente que o nome do plugin esteja prefixado antes do nome do componente. Por exemplo:

```
// Componente definido no plugin 'ContactManager'
class ExampleComponent extends Component {
}

// dentro de seu controller:
public $components = array('ContactManager.Example');
```

A mesma técnica se aplica aos Helpers e Behaviors.

Nota: Ao criar Helpers você pode notar que AppHelper não está disponível automaticamente. Você deve declarar os recursos que precisar com Uses:

```
// Declare o uso do AppHelper para seu Helper Plugin
App::uses('AppHelper', 'View/Helper');
```

Expanda seu Plugin

Este exemplo criou um bom começo para um plugin, mas há muito mais coisas que você pode fazer. Como uma regra geral, qualquer coisa que você pode fazer com sua aplicação, você pode fazer dentro de um plugin em seu lugar.

Vá em frente, inclua algumas bibliotecas de terceiros em "Vendor", adicione algumas novas shells para o cake console, e não se esqueça de criar casos de testes para que usuários de seus plugins possam testar automaticamente as funcionalidades de seus plugins!

Em nosso exemplo ContactManager, poderíamos criar os actions add/remove/edit/delete em ContactsController, implementar a validação no model Contact, e implementar uma funcionalidade que poderia se esperar ao gerenciar seus contatos. Cabe a você decidir o que implementar em seus plugins. Só não se esqueça de compartilhar seu código com a comunidade para que todos possam se beneficiar de seus impressionantes componentes reutilizáveis!

Plugin Dicas

Uma vez que o plugin foi instalado em /app/Plugin, você pode acessá-lo através da URL /plugin_name/controller_name/action. Em nosso plugin ContactManager de exemplo, acessamos nosso ContactsController com /contact_manager/contacts.

Algumas dicas finais sobre como trabalhar com plugins em suas aplicações CakePHP:

- Quando você não tiver um [Plugin]AppController e [Plugin]AppModel, você terá um erro Missing Controller quando estiver tentando acessar um controller de plugin.
- Você pode definir seus layouts para plugins, dentro de app/Plugin/[Plugin]/View/Layouts. Caso contrário, o plugin irá utilizar por padrão os layouts da pasta /app/View/Layouts.
- Você pode fazer um inter-plugin de comunicação usando `$this->requestAction('/plugin_name/controller_name/action');` em seus controllers.

- Se você usar requestAction, esteja certo que os nomes dos controllers e das models sejam tão únicos quanto possível. Caso contrário você poderá obter do PHP o erro «redefined class . . . »

CAPÍTULO 7

Desenvolvimento

Nesta seção vamos cobrir os vários aspectos do desenvolvimento de uma aplicação CakePHP. Temas como configuração, manipulação de erros e exceções, depuração e testes serão abordados.

Configuration

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[40] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Database Configuration

Routing

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[41] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[40] https://github.com/cakephp/docs
[41] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Connecting Routes

Routes Configuration

Route Elements

Passed Arguments

Named Parameters

Prefix Routing

Sessions

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[42] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Exceptions

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[43] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Error Handling

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[44] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[42] https://github.com/cakephp/docs
[43] https://github.com/cakephp/docs
[44] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Debugging

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[45] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Testing

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[46] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Running Tests

REST

Muitos programadores de aplicação mais recentes estão percebendo a necessidade de abrir sua principal funcionalidade para um público maior. Fornecer fácil, acesso irrestrito à sua API principal pode ajudar a obter sua plataforma aceito, e permite manipulações e fácil integração com outros sistemas.

Embora existam outras soluções, o REST é uma ótima maneira de facilitar acesso à lógica que você criou no seu aplicativo. Está simples, geralmente baseado em XML (falamos XML simples, nada como um Envelope SOAP) e depende de cabeçalhos HTTP para direção. Exposição Uma API via REST no CakePHP é simples.

A Configuração Simples

A maneira mais rápida de começar a funcionar com o REST é adicionar algumas linhas ao seu arquivo route.php, encontrado no app / Config. O objeto do roteador possui um método chamado `mapResources()`, que é usado para configurar uma série de rotas padrão para o acesso REST aos seus controladores. Certifique-se de que `mapResources()` vem antes de `pedir CAKE . 'Config' . DS . 'routes.php';` e outras rotas que ultrapassariam as rotas. Se quisermos permitir o acesso REST a um banco de dados de receitas, faríamos algo assim:

[45] https://github.com/cakephp/docs
[46] https://github.com/cakephp/docs

```
//In app/Config/routes.php...

Router::mapResources('recipes');
Router::parseExtensions();
```

A primeira linha configura uma série de rotas padrão para acesso REST fácil enquanto `parseExtensions()` especifica o formato de resultado desejado (por exemplo, xml, json, rss). Essas rotas são sensíveis ao método de solicitação HTTP.

HTTP format	URL format	Controller action invoked
GET	/recipes.format	RecipesController::index()
GET	/recipes/123.format	RecipesController::view(123)
POST	/recipes.format	RecipesController::add()
POST	/recipes/123.format	RecipesController::edit(123)
PUT	/recipes/123.format	RecipesController::edit(123)
DELETE	/recipes/123.format	RecipesController::delete(123)

A classe do roteador do CakePHP usa uma série de indicadores diferentes para detectar o método HTTP que está sendo usado. Aqui eles estão em ordem de preferência:

1. The *_method* POST variable
2. The X_HTTP_METHOD_OVERRIDE
3. The REQUEST_METHOD header

A variável *_method* POST é útil na utilização de um navegador como um Cliente REST (ou qualquer outra coisa que possa fazer POST facilmente). Basta configurar o valor de _method para o nome do método de solicitação HTTP você deseja emular.

Uma vez que o roteador foi configurado para mapear solicitações REST para determinados ações de controle, podemos avançar para criar a lógica em nosso ações do controlador. Um controlador básico pode parecer algo como esta:

```
// Controller/RecipesController.php
class RecipesController extends AppController {

    public $components = array('RequestHandler');

    public function index() {
        $recipes = $this->Recipe->find('all');
        $this->set(array(
            'recipes' => $recipes,
            '_serialize' => array('recipes')
        ));
    }

    public function view($id) {
        $recipe = $this->Recipe->findById($id);
        $this->set(array(
            'recipe' => $recipe,
            '_serialize' => array('recipe')
        ));
    }

    public function add() {
        $this->Recipe->create();
        if ($this->Recipe->save($this->request->data)) {
            $message = 'Saved';
```

```
        } else {
            $message = 'Error';
        }
        $this->set(array(
            'message' => $message,
            '_serialize' => array('message')
        ));
    }

    public function edit($id) {
        $this->Recipe->id = $id;
        if ($this->Recipe->save($this->request->data)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set(array(
            'message' => $message,
            '_serialize' => array('message')
        ));
    }

    public function delete($id) {
        if ($this->Recipe->delete($id)) {
            $message = 'Deleted';
        } else {
            $message = 'Error';
        }
        $this->set(array(
            'message' => $message,
            '_serialize' => array('message')
        ));
    }
}
```

Uma vez que adicionamos uma chamada para `Router::parseExtensions()`, o roteador CakePHP já está preparado para exibir diferentes visualizações com base em diferentes tipos de requests. Como estamos lidando com REST requests, estaremos fazendo visualizações XML. Você também pode facilmente fazer exibições JSON usando CakePHP's built-in *JSON and XML views*. Ao usar o built in `XmlView` podemos definir uma variável de visualização `_serialize`. Este especial A variável view é usada para definir quais variáveis de exibição `XmlView` devem serializar em XML.

Se quisermos modificar os dados antes de ser convertidos em XML, não devemos defina a variável de exibição `_serialize` e, em vez disso, use arquivos de exibição. Nós colocamos as visualizações REST para nosso RecipesController dentro de `app/View/recipes/xml`. Nós também podemos usar o `Xml` para saída XML rápida e fácil nessas vistas. Aqui é o que nossa exibição de índice pode parecer:

```
// app/View/Recipes/xml/index.ctp
// Do some formatting and manipulation on
// the $recipes array.
$xml = Xml::fromArray(array('response' => $recipes));
echo $xml->asXML();
```

Ao atender um tipo de conteúdo específico usando parseExtensions(), CakePHP procura automaticamente um auxiliar de visualização que corresponda ao tipo. Como estamos usando o XML como o tipo de conteúdo, não há um ajudante interno, no entanto, se você criasse um, ele seria automaticamente carregado para o nosso uso nessas visualizações.

O XML renderizado acabará por parecer algo assim:

```
<recipes>
    <recipe id="234" created="2008-06-13" modified="2008-06-14">
        <author id="23423" first_name="Billy" last_name="Bob"></author>
        <comment id="245" body="Yummy yummmy"></comment>
    </recipe>
    <recipe id="3247" created="2008-06-15" modified="2008-06-15">
        <author id="625" first_name="Nate" last_name="Johnson"></author>
        <comment id="654" body="This is a comment for this tasty dish."></comment>
    </recipe>
</recipes>
```

Criar a lógica para a ação de edição é um pouco mais complicado, mas não por muito. Como você está fornecendo uma API que emite XML, é um escolha natural para receber XML como entrada. Não se preocupe, o `RequestHandler` e `Router` fazem coisas muito mais fáceis. Se uma solicitação POST ou PUT tiver um tipo de conteúdo XML, então a entrada é executada através da classe CakePHP's `Xml` class, e a a representação de matriz dos dados é atribuída a *$this->request->data*. Devido a esse recurso, manipulando dados XML e POST em paralelo é sem costura: nenhuma alteração é necessária para o controlador ou o código do modelo. Tudo o que você precisa deve acabar em `$this->request->data`.

Aceitando Entrada em Outros Formatos

Normalmente, os aplicativos REST não apenas exibem conteúdo em formatos de dados alternativos, mas também aceitam dados em diferentes formatos. No CakePHP, o `RequestHandlerComponent` ajuda a facilitar isso. Por padrão, ele decodificará qualquer entrada de dados de entrada JSON/XML para pedidos POST/PUT e forneça a versão de matriz desses dados em *$this->request->data*. Você também pode filmar em desserializadores adicionais para formatos alternativos se você precisa deles, usando `RequestHandler::addInputType()`.

Modificando as rotas REST padrão

Novo na versão 2.1.

Se as rotas REST padrão não funcionarem para seu aplicativo, você pode modificá-las usando `Router::resourceMap()`. Este método permite que você defina o rotas padrão que são configuradas com `Router::mapResources()`. Ao usar esse método você precisa definir *all* os padrões que deseja usar:

```
Router::resourceMap(array(
    array('action' => 'index', 'method' => 'GET', 'id' => false),
    array('action' => 'view', 'method' => 'GET', 'id' => true),
    array('action' => 'add', 'method' => 'POST', 'id' => false),
    array('action' => 'edit', 'method' => 'PUT', 'id' => true),
    array('action' => 'delete', 'method' => 'DELETE', 'id' => true),
    array('action' => 'update', 'method' => 'POST', 'id' => true)
));
```

Ao substituir o mapa de recursos padrão, as futuras chamadas para `mapResources()` irão use os novos valores.

Roteamento REST personalizado

Se as rotas padrão criadas por `Router::mapResources()` não funciona para você, use o método `Router::connect()` para definir um conjunto personalizado de Rotas de REST. O método `connect()` permite que você defina uma série de diferentes opções para um determinado URL. Consulte a seção em route-conditions para obter mais informações.

Novo na versão 2.5.

Você pode fornecer a chave `connectOptions` na matriz ‘‘$options‘ para `Router::mapResources()` para fornecer configurações personalizadas usadas por `Router::connect()`:

```
Router::mapResources('books', array(
    'connectOptions' => array(
        'routeClass' => 'ApiRoute',
    )
));
```

Dispatcher Filters

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[47] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

---

[47] https://github.com/cakephp/docs

CAPÍTULO 8

Implementação

Uma vez que sua aplicação está completa, ou mesmo antes quando você quiser colocá-la no ar. Existem algumas poucas coisas que você deve fazer quando colocar em produção uma aplicação CakePHP.

Definindo a Raiz

Definir a raiz do documento da sua aplicação corretamente é um passo importante para manter seu código protegido e sua aplicação mais segura. As aplicações desenvolvidas com o CakePHP devem ter a raiz apontando para o diretório `app/webroot`. Isto torna a aplicação e os arquivos de configurações inacessíveis via URL. Configurar a raiz do documento depende de cada webserver. Veja a *Instalação Avançada* para informações sobre webservers específicos.

Atualizar o core.php

Atualizar o arquivo `core.php`, especificamente o valor do `debug` é de extrema importância. Tornar o debug igual a `0` desabilita muitos recursos do processo de desenvolvimento que nunca devem ser expostos ao mundo. Desabilitando o debug, as coisas a seguir mudarão:

- Mensagens de depuração criadas com `pr()` e `debug()` serão desabilitadas.
- O cache interno do CakePHP será descartado após 99 anos em vez de 10 segundos.
- *Views* de erros serão menos informativas, retornando mensagens de erros genéricas.
- Erros não serão mostrados.
- O rastro da pilha de exceções será desabilitado.

Além dos itens citados acima, muitos plugins e extensões usam o valor do `debug` para modificarem seus comportamentos.

Multiplas aplicações usando o mesmo core do CakePHP

Existem algumas maneiras de você configurar múltiplas aplicações para usarem o mesmo *core* (núcleo) do CakePHP. Você pode usar o `include_path` do PHP ou definir a constante `CAKE_CORE_INCLUDE_PATH` no `webroot/index.php` da sua aplicação. Geralmente, usar o `include_path` do PHP é a meneira mais fácil e robusta pois o CakePHP já vem pré-configurado para olhar o `include_path`.

No seu arquivo `php.ini` localize a diretiva `include_path` existente e anexe no final o caminho para o diretório `lib` do CakePHP:

```
include_path = '.:/usr/share/php:/usr/share/cakephp-2.0/lib'
```

Assumimos que você está rodando um servidor *nix e instalou o CakePHP em `/usr/share/cakephp-2.0`.

CAPÍTULO 9

Tutoriais & Exemplos

Nesta seção, você poderá caminhar ao longo de típicas aplicações CakePHP para ver como todas as peças se encaixam.

Como alternativa, você pode preferir visitar o repositório não oficial de plugins para o CakePHP CakePackages[48] ou a Padaria[49] para conhecer aplicações e componentes existentes.

Blog

Bem vindo ao CakePHP. Você provavelmente está lendo este tutorial porque quer aprender mais sobre como o CakePHP funciona. Nosso objetivo é aumentar a produtividade e fazer a programação uma tarefa mais divertida: Esperamos que você veja isto na prática enquanto mergulha nos códigos.

Este tutorial irá cobrir a criação de uma aplicação de blog simples. Nós iremos baixar e instalar o Cake, criar e configurar o banco de dados e criar a lógica da aplicação suficiente para listar, adicionar, editar e deletar posts do blog.

Aqui vai uma lista do que você vai precisar:

1. Um servidor web rodando. Nós iremos assumir que você esteja usando o Apache, embora as instruções para usar outros servidores sejam bem semelhantes. Talvez tenhamos que brincar um pouco com as configurações do servidor mas a maioria das pessoas serão capazes de ter o Cake rodando sem precisar configurar nada.
2. Um servidor de banco de dados. Nós iremos usar o MySQL Server neste tutorial. Você precisa saber o mínimo sobre SQL para criar um banco de dados. O Cake pegará as rédeas a partir deste ponto.
3. Conhecimento básico da linguagem PHP. Quanto mais orientado a objetos você já programou, melhor: mas não tenha medo se é fã de programação procedural.
4. E por último, você vai precisar de um conhecimento básico do padrão de projetos MVC. Uma rápida visão geral pode ser encontrada em *Entendendo o Model-View-Controller*. Não se preocupe, deve ter meia página ou menos.

[48] https://plugins.cakephp.org/
[49] https://bakery.cakephp.org/

Então, vamos começar!

Baixando o Cake

Primeiro, vamos baixar uma cópia recente do CakePHP.

Para fazer o download de uma cópia recente, visite o projeto do CakePHP no github: https://github.com/cakephp/cakephp/downloads e faça o download da última versão 2.0.

Você também pode clonar o repositório usando o git[50]. `git clone git://github.com/cakephp/cakephp.git`.

Idependente da maneira de como você baixou o Cake, coloque o código obtido dentro do seu diretório web público. A estrutura dos diretórios deve ficar parecido com o seguinte:

```
/caminho_para_diretorio_web_publico
    /app
    /lib
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

Agora pode ser um bom momento para aprender um pouco sobre como funciona a estrutura de diretórios do CakePHP: Veja a seção *Estrutura de Diretórios no CakePHP*.

Criando o Banco de Dados do Blog

Em seguida, vamos configurar o banco de dados correspondente ao nosso blog. Se você já não tiver feito isto, crie um banco de dados vazio para usar neste tutorial com o nome que desejar. Neste momento, vamos criar apenas uma tabela para armazenar nossos posts. Também vamos inserir alguns posts para usar como teste. Execute as instruções a seguir no seu banco de dados:

```
-- Primeiro, criamos nossa tabela de posts
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

-- Agora inserimos alguns posts para testar
INSERT INTO posts (title, body, created)
    VALUES ('The title', 'This is the post body.', NOW());
INSERT INTO posts (title, body, created)
    VALUES ('A title once again', 'And the post body follows.', NOW());
INSERT INTO posts (title, body, created)
    VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

A escolha do nome de tabelas e colunas não são arbitrárias. Se você seguir as convenções de nomenclatura para estruturas do banco de dados e as convenções para nomes de classes (ambas descritas em *Convenções no CakePHP*), você será capaz de tirar proveito de muitas funcionalidades do CakePHP e evitar arquivos de configurações. O Cake

[50] http://git-scm.com/

é flexivel o bastante para acomodar até mesmo os piores esquemas de banco de dados legados, mas aderindo as convenções você poupa seu tempo.

Veja *Convenções no CakePHP* para mais informações. Aqui, basta dizer que ao nomear nossa tabela de "posts", automaticamente ela será ligada ao nosso model Post e as colunas "modified" e "created" serão «automagicamente» atualizadas pelo CakePHP.

Configurações do Banco de Dados

Para o Alto e Avante: Vamos agora avisar ao Cake onde está nosso banco de dados e como conectar a ele. Para muitos, esta é a primeira e última configuração a ser feita.

Uma exemplo de arquivo de configuração do banco de dados pode ser encontrado em `/app/Config/database.php.default`. Copie este arquivo no mesmo diretório renomeando-o para `database.php`.

O arquivo é bem simples: basta alterar os valores da variável $default com os dados da nossa configuração. Um exemplo completo desta configuração irá se parecer com esta:

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

Após salvar seu novo arquivo database.php, você estará apto para abrir seu navegador e ver a página de boas vindas do Cake. A página de boas vindas deverá lhe mostrar uma mensagem dizendo que seu arquivo de conexão com o banco de dados foi encontrado, e que o Cake conseguiu se conectar com o banco de dados.

Configuração Opcional

Existem outros três itens que podem ser configurados. Muitos desenvolvedores sempre configuram estes itens, mas eles não são obrigatórios para este tutorial. Uma das configurações é customizar uma string (ou «salt») para ser utilizada nos hashes de segurança. O segundo é definir um número (ou «seed») para uso em criptografia. E o terceiro é dar permissão de escrita para o CakePHP na pasta `tmp`.

O «security salt» é utilizado para gerar hashes. Altere o valor padrão do salt editando o arquivo `/app/Config/core.php` na linha 187. Não importa muito o que o novo valor seja, basta que não seja fácil de adivinhar.

```
/**
 * A random string used in security hashing methods.
 */
Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
?>
```

O «cipher seed» é usado para criptografar/descriptografar strings. Altere o valor padrão editando o arquivo `/app/Config/core.php` na linha 192. Como no «security salt», não importa muito o que o novo valor seja, basta que não seja fácil de adivinhar.

```
/**
 * A random numeric string (digits only) used to encrypt/decrypt strings.
 */
Configure::write('Security.cipherSeed', '7485712659625147843639846751');
?>
```

A última tarefa é garantir acesso de escrita para a pasta `app/tmp`. A melhor maneira para fazer isto é localizar o usuário com que o seu servidor web é executado (`<?php echo `whoami`; ?>`) e alterar o dono da pasta `app/tmp` para este usuário. Você pode executar (em *nix) o comando a seguir para alterar o usuário dono da pasta.

```
$ chown -R www-data app/tmp
```

Se por alguma razão o CakePHP não conseguir escrever nesta pasta, você será avisado por uma mensagem enquanto estiver em modo de desenvolvimento.

Uma Palavra Sobre o mod_rewrite

Ocasionalmente, um novo usuário irá esbarrar em problemas com o mod_rewrite, então vou abordá-los superficialmente aqui. Se a página de boas-vindas do CakePHP parecer um pouco sem graça (sem imagens, sem cores e sem os estilos css), isso é um indício de que o mod_rewrite provavelmente não esteja funcionando em seu sistema. Aqui estão algumas dicas para lhe ajudar a deixar tudo funcionando corretamente:

1. Certifique-se de que a sobrescrita de opções do .htaccess está habilitada: em seu arquivo httpd.conf, você deve ter uma parte que define uma seção para cada <Directory> do seu servidor. Certifique-se de que a opção `AllowOverride` esteja com o valor `All` para o <Directory> correto. Por questões de segurança e performance, *não* defina `AllowOverride` para `All` em `<Directory />`. Ao invés disso, procure o bloco `<Directory>` que se refere ao seu diretório raíz de seu website.
2. Certifique-se de estar editando o arquivo httpd.conf ao invés de algum específico, que seja válido apenas para um dado usuário ou para um dado site.
3. Por alguma razão, você pode ter obtido uma cópia do CakePHP sem os arquivos .htaccess. Isto algumas vezes acontece porque alguns sistemas operacionais tratam arquivos que começam com "." como arquivos ocultos e normalmente não fazem cópias deles. Certifique-se de obter sua cópia do CakePHP diretamente da seção de downloads do site ou de nosso repositório git.
4. Certifique-se de que o Apache esteja carregando o mod_rewrite corretamente! Você deve ver algo como:

   ```
   LoadModule rewrite_module       libexec/httpd/mod_rewrite.so
   ```

   ou (para o Apache 1.3):

   ```
   AddModule                       mod_rewrite.c
   ```

   em seu httpd.conf.

Se você não quiser ou não puder carregar o mod_rewrite (ou algum outro módulo compatível) em seu servidor, você vai precisar usar o recurso de URLs amigáveis do CakePHP. No arquivo `/app/Config/core.php`, descomente uma linha parecida com:

```
Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

E remova também os arquivos .htaccess em:

```
/.htaccess
/app/.htaccess
/app/webroot/.htaccess
```

Com isto, suas URLs ficarão parecidas com www.exemplo.com/index.php/nomecontroller/nomeaction/param ao invés de www.exemplo.com/nomecontroller/nomeaction/param.

Se você está instalando o CakePHP em outro webserver diferente do Apache, você pode encontrar instruções para ter a reescrita de URLs funcionando na seção *Instalação Avançada*.

Continue lendo este tutorial em *Blog - Continuação* para começar a construir sua primeira aplicação CakePHP.

Blog - Continuação

Crie um Model Post

A classe Model é o pão com manteiga das aplicações CakePHP. Ao criar um model do CakePHP que irá interagir com nossa base de dados, teremos os alicerces necessários para posteriormente fazer nossas operações de visualizar, adicionar, editar e excluir.

Os arquivos de classe do tipo model do CakePHP ficam em `/app/Model` e o arquivo que iremos criar será salvo em `/app/Model/Post.php`. O conteúdo completo deste arquivo deve ficar assim:

```
class Post extends AppModel {
    public $name = 'Post';
}
```

A nomenclatura da classe segue uma convenção que é muito importante no CakePHP. Ao chamar nosso model de Post, o CakePHP pode automaticamente deduzir que este model será usado num PostsController, e que manipulará os dados de uma tabela do banco chamada de `posts`.

Nota: O CakePHP irá criar um objeto (instância) do model dinamicamente para você, se não encontrar um arquivo correspondente na pasta /app/Model. Isto também significa que, se você acidentalmente der um nome errado ao seu arquivo (p.ex. post.php ou posts.php) o CakePHP não será capaz de reconhecer nenhuma de suas configurações adicionais e ao invés disso, passará a usar seus padrões definidos internamente na classe Model.

Para saber mais sobre models, como prefixos de nomes de tabelas, callbacks e validações, confira o capítulo sobre /models deste manual.

Crie o Controller Posts

A seguir, vamos criar um controller para nossos posts. O controller é onde toda a lógica de negócio para interações vai acontecer. De uma forma geral, é o local onde você vai manipular os models e lidar com o resultado das ações feitas sobre nossos posts. Vamos pôr este novo controller num arquivo chamado `PostsController.php` dentro do diretório `/app/Controller`. Aqui está como um controller básico deve se parecer:

```
class PostsController extends AppController {
    public $helpers = array ('Html','Form');
    public $name = 'Posts';
}
```

Agora, vamos adicionar uma action ao nosso controller. Actions quase sempre representam uma única função ou interface numa aplicação. Por exemplo, quando os usuários acessarem o endereço www.exemplo.com/posts/index (que, neste caso é o mesmo que www.exemplo.com/posts/), eles esperam ver a listagem dos posts. O código para tal ação deve se parecer com algo assim:

```
class PostsController extends AppController {
    public $helpers = array ('Html','Form');
    public $name = 'Posts';

    function index() {
        $this->set('posts', $this->Post->find('all'));
    }
}
```

Deixe-me explicar a ação um pouco. Definindo a função `index()` em nosso PostsController, os usuários podem acessar esta lógica visitando o endereço www.exemplo.com/posts/index. De maneira semelhante, se definirmos um método chamado `foobar()` dentro do controller, os usuários deveriam ser capazes de acessá-lo pelo endereço www.exemplo.com/posts/foobar.

Aviso: Você pode ficar tentado a nomear seus controller e actions de uma certa maneira visando obter uma certa URL. Mas resista a esta tentação. Siga as convenções do CakePHP (nomes de controllers no plural, etc) e crie nomes de actions e controllers que sejam legíveis e também compreensíveis. Você sempre vai poder mapear URLs para seu código utilizando «rotas», conforme mostraremos mais à frente.

A única declaração na nossa action utiliza o método `set()` para passar dados do controller para a view (que vamos criar logo mais). A linha define uma variável na view chamada "posts" que vai conter o retorno da chamada do método `find('all')` do model Post. Nosso model Post está automaticamente disponível como `$this->Post` uma vez que seguimos as convenções de nomenclatura do Cake.

Para aprender mais sobre controllers do CakePHP, confira a seção *Controllers*.

Criando as Views de Posts

Agora que temos nossos dados chegando ao nosso model e com a lógica da nossa aplicação definida em nosso controller, vamos criar uma view para a ação `index()` que criamos acima.

As views do Cake são meros fragmentos voltados à apresentação de dados que vão dentro do layout da aplicação. Para a maioria das aplicações, as views serão marcações HTML intercalados com código PHP, mas as views também podem ser renderizadas como XML, CVS ou mesmo como dados binários.

Os layouts são páginas que encapsulam as views e que podem ser intercambiáveis, mas por agora, vamos apenas usar o layout padrão.

Lembra da última seção, em que associamos a variável "posts" para a view usando o método `set()`? Com aquilo, os dados foram repassados para a view num formato parecido com este:

```
// print_r($posts) output:

Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => The title
                    [body] => This is the post body.
                    [created] => 2008-02-13 18:34:55
                    [modified] =>
                )
```

```
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => A title once again
                    [body] => And the post body follows.
                    [created] => 2008-02-13 18:34:56
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Title strikes back
                    [body] => This is really exciting! Not.
                    [created] => 2008-02-13 18:34:57
                    [modified] =>
                )
        )
)
```

Os arquivos de view do Cake são armazenados na pasta `/app/View` dentro de uma pasta com o mesmo nome do controller a que correspondem (em nosso caso, vamos criar uma pasta chamada "Posts"). Para apresentar os dados do post num formato adequado de tabela, o código de nossa view deve ser algo como:

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Posts do Blog</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Título</th>
        <th>Data de Criação</th>
    </tr>

    <!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo as informações␣
↪dos posts -->

    <?php foreach ($posts as $post): ?>
        <tr>
            <td><?php echo $post['Post']['id']; ?></td>
            <td>
                <?php echo $this->Html->link($post['Post']['title'],
                    array('controller' => 'posts', 'action' => 'view', $post['Post'][
↪'id'])); ?>
            </td>
            <td><?php echo $post['Post']['created']; ?></td>
        </tr>
    <?php endforeach; ?>

</table>
```

Isto é tão simples quanto parece!

Você deve ter notado o uso de um objeto chamado `$this->Html`. Esta é uma instância da classe `HtmlHelper` do CakePHP. O CakePHP vem com um conjunto de helpers que tornam uma moleza fazer coisas como criar links, gerar formulários, Javascript e elementos dinâmicos com Ajax. Você pode aprender mais sobre como usá-los na seção *Helpers*, mas o importante a ser notado aqui é que o método `link()` irá gerar um link em HTML com o título (o primeiro parâmetro) e URL (o segundo parâmetro) dados.

Ao especificar URLs no Cake, é recomendado que você use o formato de array. Este assunto é explicado com mais detalhes na seção sobre Rotas. Usar o formato de array para URLs, permite que você tire vantagens da capacidade do CakePHP de reverter este formato de URL em URLs relativas e vice versa. você também pode simplesmente informar um caminho relativo à base da aplicação na forma /controller/action/parametro_1/parametro_2.

Neste ponto, você deve ser capaz de apontar seu navegador para http://www.exemplo.com/posts/index. Você deve ver sua view, corretamente formatada com o título e a tabela listando os posts.

Se lhe ocorreu clicar num dos links que criamos nesta view (no título do post e que apontam para uma URL /posts/view/algum_id), você provavelmente recebeu uma mensagem do CakePHP dizendo que a action ainda não foi definida. Se você não tiver visto um aviso assim, então ou alguma coisa deu errado ou então você já tinha definido uma action anteriormente, e neste caso, você é muito afoito. Se não, vamos criá-la em nosso PostsController agora:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
    public $name = 'Posts';

    public function index() {
         $this->set('posts', $this->Post->find('all'));
    }

    public function view($id = null) {
        $this->set('post', $this->Post->findById($id));
    }
}
```

A chamada do método `set()` deve lhe parece familiar. Perceba que estamos usando o método `read()` ao invés do `find('all')` porque nós realmente só queremos informações de um único post.

Note que a action de nossa view recebe um parâmetro: O ID do post que queremos ver. Este parâmetro é repassado à action por meio da URL requisitada. Se um usuário acessar uma URL /posts/view/3, então o valor "3" será atribuído ao parâmetro `$id`.

Agora vamos criar a view para nossa nova action "view" e colocá-la em `/app/View/Posts/view.ctp`:

```
<!-- File: /app/View/Posts/view.ctp -->

<h1><?php echo $post['Post']['title']?></h1>

<p><small>Created: <?php echo $post['Post']['created']?></small></p>

<p><?php echo $post['Post']['body']?></p>
```

Confira se está funcionando tentando acessar os links em `/posts/index` ou requisitando diretamente um post acessando `/posts/view/1`.

Adicionando Posts

Ler a partir da base de dados e exibir os posts foi um grande começo, mas precisamos permitir também que os usuários adicionem novos posts.

Primeiramente, comece criando uma action `add()` no PostsController:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form', 'Flash');
    public $components = array('Flash');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }

    public function view($id) {
        $this->set('post', $this->Post->findById($id));
    }

    public function add() {
        if ($this->request->is('post')) {
            if ($this->Post->save($this->request->data)) {
                $this->Flash->success('Your post has been saved.');
                $this->redirect(array('action' => 'index'));
            }
        }
    }
}
```

Nota: Você precisa incluir o componente FlashComponent e o helper FlashHelper em qualquer controller que você manipula variáveis de sessão. Neste caso, incluímos apenas o componente porque ele carrega o helper automaticamente. Se você sempre utiliza sessões, inclua o componente no seu arquivo AppController.

Aqui está o que a action `add()` faz: se o método da requisição feita pelo cliente for do tipo post, ou seja, se ele enviou dados pelo formulário, tenta salvar os dados usando o model Post. Se, por alguma razão ele não salvar, apenas renderize a view. Isto nos dá uma oportunidade de mostrar erros de validação e outros avisos ao usuário.

Quando um usuário utiliza um formulário para submeter (POSTar) dados para sua aplicação, esta informação fica disponível em `$this->request->data`.Você pode usar as funções `pr()` ou `debug()` para exibir os dados se você quiser conferir como eles se parecem.

Nós usamos o método `FlashComponent::success()` do componente FlashComponent para definir uma variável de sessão com uma mensagem a ser exibida na página depois de ser redirecionada. No layout, nós temos `FlashHelper::render()` que exibe a mensagem e limpa a variável de sessão correspondente. O método `Controller::redirect` do controller redireciona para outra URL. O parâmetro `array('action' => 'index')` é convertido para a URL /posts, em outras palavras, a action index do controller posts. Você pode conferir a função `Router::url()` na API para ver os formatos que você pode usar ao especificar uma URL para actions do CakePHP.

Chamar o método `save()` irá verificar por erros de validação e abortar o salvamento se algum erro ocorrer. Vamos falar mais sobre erros de validação e sobre como manipulá-los nas seções seguintes.

Validação de Dados

O CakePHP percorreu uma longa estrada combatendo a monotonia da validação de dados de formulários. Todo mundo detesta codificar formulários intermináveis e suas rotinas de validação. O CakePHP torna tudo isso mais fácil e mais rápido.

Para usufruir das vantagens dos recursos de validação, você vai precisar usar o FormHelper do Cake em suas views. O `FormHelper` está disponível por padrão em todas as suas views na variável `$this->Form`.

Aqui está nossa view add:

```
<!-- File: /app/View/Posts/add.ctp -->

<h1>Add Post</h1>
<?php
echo $this->Form->create('Post');
echo $this->Form->input('title');
echo $this->Form->input('body', array('rows' => '3'));
echo $this->Form->end('Save Post');
```

Aqui, usamos o FormHelper para gerar a tag de abertura para um formulário. Aqui está o HTML gerado pelo `$this->Form->create()`:

```
<form id="PostAddForm" method="post" action="/posts/add">
```

Se o método `create()` for chamado sem quaisquer parâmetros, o CakePHP assume que você está criando um formulário que submete para a action `add()` do controller atual (ou para a action `edit()` se um campo id for incluído nos dados do formulário), via POST.

O método `$this->Form->input()` é usado para criar elementos de formulário de mesmo nome. O primeiro parâmetro informa ao CakePHP qual o campo correspondente e o segundo parâmetro permite que você especifique um extenso array de opções. Neste caso, o número de linhas para o textarea. Há alguma introspecção «automágica» envolvida aqui: o `input()` irá exibir diferentes elementos de formulário com base no campo do model em questão.

A chamada à `$this->Form->end()` gera um botão de submissão e encerra o formulário. Se uma string for informada como primeiro parâmetro para o `end()`, o FormHelper exibe um botão de submit apropriadamente rotulado junto com a tag de fechamento do formulário. Novamente, confira o capítulo sobre os *Helpers* disponíveis no CakePHP para mais informações sobre os helpers.

Agora vamos voltar e atualizar nossa view `/app/View/Post/index.ctp` para incluir um novo link para «Adicionar Post». Antes de <table>, adicione a seguinte linha:

```
echo $this->Html->link('Add Post', array('controller' => 'posts', 'action' => 'add'));
```

Você pode estar imaginando: como eu informo ao CakePHP sobre os requisitos de validação de meus dados? Regras de validação são definidas no model. Vamos olhar de volta nosso model Post e fazer alguns pequenos ajustes:

```
class Post extends AppModel {
    public $name = 'Post';

    public $validate = array(
        'title' => array(
            'rule' => 'notEmpty'
        ),
        'body' => array(
            'rule' => 'notEmpty'
        )
    );
}
```

O array `$validate` diz ao CakePHP sobre como validar seus dados quando o método `save()` for chamado. Aqui, eu especifiquei que tanto os campos body e title não podem ser vazios. O mecanismo de validação do CakePHP é robusto, com diversas regras predefinidas (números de cartão de crédito, endereços de e-mail, etc.) além de ser bastante flexível, permitindo adicionar suas próprias regras de validação. Para mais informações, confira o capítulo sobre /models/data-validation.

Agora que você incluiu as devidas regras de validação, tente adicionar um post com um título ou com o corpo vazio para ver como funciona. Uma vez que usamos o método `FormHelper::input()` do FormHelper para criar nossos elementos de formulário, nossas mensagens de erros de validação serão mostradas automaticamente.

Editando Posts

Edição de Posts: Aqui vamos nós. A partir de agora você já é um profissional do CakePHP, então você deve ter identificado um padrão. Criar a action e então criar a view. Aqui está como o código da action `edit()` do PostsController deve se parecer:

```
function edit($id = null) {
    $this->Post->id = $id;
    if ($this->request->is('get')) {
        $this->request->data = $this->Post->findById($id);
    } else {
        if ($this->Post->save($this->request->data)) {
            $this->Flash->success('Your post has been updated.');
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

Esta action primeiro verifica se a requisição é do tipo GET. Se for, nós buscamos o Post e passamos para a view. Se a requisição não for do tipo GET, provavelmente esta contém dados de um formulário POST. Nós usaremos estes dados para atualizar o registro do nosso Post ou exibir novamente a view mostrando para o usuário os erros de validação.

A view edit pode ser algo parecido com isto:

```
<!-- File: /app/View/Posts/edit.ctp -->

<h1>Edit Post</h1>
<?php
    echo $this->Form->create('Post', array('action' => 'edit'));
    echo $this->Form->input('title');
    echo $this->Form->input('body', array('rows' => '3'));
    echo $this->Form->input('id', array('type' => 'hidden'));
    echo $this->Form->end('Save Post');
```

Esta view exibe o formulário de edição (com os valores populados), juntamente com quaisquer mensagens de erro de validação.

Uma coisa a atentar aqui: o CakePHP vai assumir que você está editando um model se o campo "id" estiver presente no array de dados. Se nenhum "id" estiver presente (como a view add de inserção), o Cake irá assumir que você está inserindo um novo model quando o método `save()` for chamado.

Você agora pode atualizar sua view index com os links para editar os posts específicos:

```
<!-- File: /app/View/Posts/index.ctp  (links para edição adicionados) -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link("Add Post", array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Action</th>
        <th>Created</th>
    </tr>

<!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo
as informações dos posts -->

<?php foreach ($posts as $post): ?>
```

```
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' =>
↪'view', $post['Post']['id']));?>
        </td>
        <td>
            <?php echo $this->Form->postLink(
                'Delete',
                array('action' => 'delete', $post['Post']['id']),
                array('confirm' => 'Are you sure?')
            )?>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post
↪']['id']));?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
<?php endforeach; ?>

</table>
```

Deletando Posts

A seguir, vamos criar uma maneira para os usuários excluírem posts. Comece com uma action `delete()` no PostsController:

```
function delete($id) {
    if (!$this->request->is('post')) {
        throw new MethodNotAllowedException();
    }
    if ($this->Post->delete($id)) {
        $this->Flash->success('The post with id: ' . $id . ' has been deleted.');
        $this->redirect(array('action' => 'index'));
    }
}
```

Esta lógica exclui o post dado por $id, e utiliza `$this->Flash->success()` para mostrar uma mensagem de confirmação para o usuário depois de redirecioná-lo para `/posts`.

Se o usuário tentar deletar um post usando uma requisição do tipo GET, nós lançamos uma exceção. Exceções não apanhadas são capturadas pelo manipulador de exceções do CakePHP e uma página de erro amigável é mostrada. O CakePHP vem com muitas *Exceptions* que você pode usar para indicar vários tipos de erros HTTP que sua aplicação pode precisar gerar.

Como estamos executando apenas uma lógica de negócio e redirecionando, esta action não tem uma view. Você pode querer atualizar sua view index com links que permitam ao usuários excluir posts, porém, como um link executa uma requisição do tipo GET, nossa action irá lançar uma exceção. Precisamos então criar um pequeno formulário que enviará um método POST adequado. Para estes casos o helper FormHelper fornece o método `postLink()`:

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link('Add Post', array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
```

```
        <th>Title</th>
        <th>Actions</th>
        <th>Created</th>
    </tr>

<!-- Aqui é onde nós percorremos nossa matriz $posts, imprimindo as informações dos
↪posts -->

    <?php foreach ($posts as $post): ?>
        <tr>
            <td><?php echo $post['Post']['id']; ?></td>
            <td>
                <?php echo $this->Html->link($post['Post']['title'], array('action'
↪ 'view', $post['Post']['id']));?>
            </td>
            <td>
                <?php echo $this->Form->postLink(
                    'Delete',
                    array('action' => 'delete', $post['Post']['id']),
                    array('confirm' => 'Are you sure?'));
                ?>
            </td>
            <td><?php echo $post['Post']['created']; ?></td>
        </tr>
    <?php endforeach; ?>

</table>
```

Nota: O código desta view também utiliza o HtmlHelper para solicitar uma confirmação do usuário com um diálogo em Javascript antes de tentar excluir o post.

Rotas

Para alguns, o roteamento padrão do CakePHP funcionará muito bem. Os desenvolvedores que estiverem mais afeitos a criar produtos ainda mais amigáveis aos usuários e aos mecanismos de busca irão gostar da maneira que as URLs do CakePHP são mapeadas para actions específicas. Então vamos fazer uma pequena alteração de rotas neste tutorial.

Para mais informações sobre técnicas avançadas de roteamento, veja *Routes Configuration*.

Por padrão, o CakePHP responde a requisições para a raiz de seu site (i.e. http://www.exemplo.com) usando seu PagesController e renderizando uma view chamada de «home». Ao invés disso, vamos substituir isto por nosso PostsController criando uma regra de roteamento.

As rotas do Cake são encontrada no arquivo `/app/Config/routes.php`. Você vai querer comentar ou remover a linha que define a rota raiz padrão. Ela se parece com:

```
Router::connect('/', array('controller' => 'pages', 'action' => 'display', 'home'));
```

Esta linha conecta a URL "/" com a home page padrão do CakePHP. Queremos conectá-la com nosso próprio controller, então adicionamos uma linha parecida com isto:

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

Isto deve conectar as requisições de "/" à action `index()` que criaremos em nosso PostsController.

Nota: O CakePHP também faz uso do "roteamento reverso" - se, com a rota definida acima, você passar `array('controller' => 'posts', 'action' => 'index')` a um método que espere um array, a URL resultante será "/". É sempre uma boa ideia usar arrays para URLs, já que é a partir disto que suas rotas definem para onde suas URLs apontam, além de garantir que os links sempre apontem para o mesmo lugar.

Conclusão

Criar aplicações desta maneira irá lhe trazer paz, honra, amor e dinheiro além de satisfazer às suas mais ousadas fantasias. Simples, não? Tenha em mente que este tutorial foi muito básico. O CakePHP possui *muito* mais recursos a oferecer e é flexível de tantas maneiras que não conseguimos mostrar aqui por questões de simplicidade. Utilize o resto deste manual como guia para construir mais aplicações ricas em recursos.

Agora que você criou uma aplicação básica com o Cake, você está pronto para a coisa real. Comece seu próprio projeto, leia o restante do Manual e da API[51].

E se você precisar de ajuda, nos vemos no canal #cakephp (e no #cakephp-pt). Seja bem-vindo ao CakePHP!

Leitura Recomendada

Estas são as tarefas comuns que pessoas aprendendo o CakePHP geralmente querem estudar:

1. *Layouts*: Customizando o layout do seu website
2. *Elements* Incluindo e reutilizando trechos de código
3. *Scaffolding (arcabouços)*: Prototipando antes de programar
4. /console-and-shells/code-generation-with-bake Gerando código CRUD básico
5. *Autenticação simples e Autorização da Aplicação*: Tutorial de autenticação e autorização de usuários

Autenticação simples e Autorização da Aplicação

Seguindo com nosso exemplo do *Blog*, imagine que queremos fornececer acesso seguro às nossas urls, baseada em autenticação de usuário. Nós também temos outro requisito, permitir que muitos autores possam criar seus próprios posts, editar e deletar os post deles sem que afete o que os outros autores fizeram em seus posts.

Criando a tabela de usuários

Primeiro, vamos criar uma nova tabela na nossa base de dados do blog para armazenar os dados de usuários:

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

[51] https://api.cakephp.org/2.x/

Nós respeitamos as convenções do CakePHP para nomear tabelas, mas também aproveitamos outra convenção: usando as colunas username e password em nossa tabela users, o CakePHP será capaz de auto configurar as coisas quando implementarmos os mecanismos de login de nossos usuários.

A próxima etapa é criar o nosso model User, responsável pelas pesquisas, gravações e validações de dados dos usuários:

```
// app/Model/User.php
class User extends AppModel {
    public $name = 'User';
    public $validate = array(
        'username' => array(
            'required' => array(
                'rule' => array('notEmpty'),
                'message' => 'A username is required'
            )
        ),
        'password' => array(
            'required' => array(
                'rule' => array('notEmpty'),
                'message' => 'A password is required'
            )
        ),
        'role' => array(
            'valid' => array(
                'rule' => array('inList', array('admin', 'author')),
                'message' => 'Please enter a valid role',
                'allowEmpty' => false
            )
        )
    );
}
```

Vamos criar também nosso UsersController, o conteúdo a seguir corresponde à classe básica UsersController gerada através da ferramenta de geração de códigos (Bake) presente no CakePHP:

```
// app/Controller/UsersController.php
class UsersController extends AppController {


    public function beforeFilter() {
        parent::beforeFilter();
        $this->Auth->allow('add', 'logout');
    }

    public function index() {
        $this->User->recursive = 0;
        $this->set('users', $this->paginate());
    }

    public function view($id = null) {
        if (!$this->User->exists($id)) {
            throw new NotFoundException(__('Invalid user'));
        }
        $this->set('user', $this->User->findById($id));
    }

    public function add() {
        if ($this->request->is('post')) {
            $this->User->create();
```

```
            if ($this->User->save($this->request->data)) {
                $this->Flash->success(__('The user has been saved'));
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Flash->error(__('The user could not be saved. Please, try␣
↪again.'));
            }
        }
    }

    public function edit($id = null) {
        $this->User->id = $id;
        if (!$this->User->exists()) {
            throw new NotFoundException(__('Invalid user'));
        }
        if ($this->request->is('post') || $this->request->is('put')) {
            if ($this->User->save($this->request->data)) {
                $this->Flash->success(__('The user has been saved'));
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Flash->error(__('The user could not be saved. Please, try␣
↪again.'));
            }
        } else {
            $this->request->data = $this->User->findById($id);
            unset($this->request->data['User']['password']);
        }
    }

    public function delete($id = null) {
        if (!$this->request->is('post')) {
            throw new MethodNotAllowedException();
        }
        $this->User->id = $id;
        if (!$this->User->exists()) {
            throw new NotFoundException(__('Invalid user'));
        }
        if ($this->User->delete()) {
            $this->Flash->success(__('User deleted'));
            $this->redirect(array('action' => 'index'));
        }
        $this->Flash->error(__('User was not deleted'));
        $this->redirect(array('action' => 'index'));
    }
```

Da mesma forma criamos as views para nossos posts no blog ou usando a ferramanta de geração de código, nós implementamos as views. Para o propósito de nosso tutorial, iremos mostrar somente o add.ctp:

```
<!-- app/View/Users/add.ctp -->
<div class="users form">
<?php echo $this->Form->create('User');?>
    <fieldset>
        <legend><?php echo __('Add User'); ?></legend>
        <?php echo $this->Form->input('username');
        echo $this->Form->input('password');
        echo $this->Form->input('role', array(
            'options' => array('admin' => 'Admin', 'author' => 'Author')
        ));
```

```
    ?>
    </fieldset>
<?php echo $this->Form->end(__('Submit'));?>
</div>
```

Autorização (login e logout)

Nós agora estamos prontos para adicionar a camada de autorização. No CakePHP isso é feito pela `AuthComponent`, uma classe responsável por solicitar login para certas ações, manipulando sign-in e sign-out, e também autorizando usuários logados a acessarem actions as quais possuem permissão.

Para adicionar esse componente em sua aplicação abra seu arquivo `app/Controller/AppController.php` e adicione as seguintes linhas:

```
// app/Controller/AppController.php
class AppController extends Controller {
    //...

    public $components = array(
        'Flash',
        'Auth' => array(
            'loginRedirect' => array('controller' => 'posts', 'action' => 'index'),
            'logoutRedirect' => array('controller' => 'pages', 'action' => 'display',
↪'home')
        )
    );

    function beforeFilter() {
        $this->Auth->allow('index', 'view');
    }
    //...
}
```

Aqui não há muito para configurar, como nós usamos convenções na tabela users. Nós somente configuramos as urls que serão carregadas após as ações de login e logout, em nosso caso são `/posts/` e `/` respectivamente.

O que fizemos na função `beforeFilter` foi dizer ao AuthComponent para não solicitar um login para todas as actions `index` e `view`, em todos os controller. Nós queremos que nossos visitantes possam ler qualquer post sem precisar se registrar no site.

Agora, nós precisamos autorizar que novos usuários possam se registrar, salvando o nome de usuário e a senha deles, e o mais importante encriptar a senha pra que ela não seja armazenada como texto plano em nosso banco de dados. Vamos dizer ao AuthComponet para permitir que usuários deslogados acessem a função add e implementar a ação de login e logout:

```
// app/Controller/UsersController.php

public function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allow('add'); // Permitindo que os usuários se registrem
}

public function login() {
    if ($this->Auth->login()) {
        $this->redirect($this->Auth->redirect());
    } else {
        $this->Flash->error(__('Invalid username or password, try again'));
```

```
    }
}

public function logout() {
    $this->redirect($this->Auth->logout());
}
```

Hash da senha não foi feito ainda, abra o seu arquivo de model `app/Model/User.php` e adicione o seguinte:

```
// app/Model/User.php
App::uses('AuthComponent', 'Controller/Component');
class User extends AppModel {

// ...

public function beforeSave($options = array()) {
    if (isset($this->data[$this->alias]['password'])) {
        $this->data[$this->alias]['password'] = AuthComponent::password($this->data[
↪$this->alias]['password']);
    }
    return true;
}

// ...
```

Então, agora toda vez que um usuário for salvo, a senha será encriptada usando o hashing padrão disponibilizado pela classe AuthComponent. Está faltando somente um arquivo view para a função de login, Aqui está ele:

```
<div class="users form">
<?php echo $this->Flash->render('auth'); ?>
<?php echo $this->Form->create('User');?>
    <fieldset>
        <legend><?php echo __('Please enter your username and password'); ?></legend>
        <?php echo $this->Form->input('username');
        echo $this->Form->input('password');
    ?>
    </fieldset>
<?php echo $this->Form->end(__('Login'));?>
</div>
```

Você pode registrar um novo usuário acessando a url `/users/add` e autenticar com as credenciais do usuário recém criado indo para a url `/users/login`. Tente também acessar qualquer outra url sem que a permisão tenha sido explicitada, como em `/posts/add`, você verá que a aplicação irá redirecioná-lo automaticamente para a página de login.

E é isso! Parece simples demais para ser verdade. Vamos voltar um pouco para explicar o que aconteceu. A função `beforeFilter` está falando para o AuthComponent não solicitar um login para a ação `add` em adição as ações `index` e `view` que foram prontamente autorizadas na função `beforeFilter` do AppController.

A ação de `login` chama a função `$this->Auth->login()` do AuthComponent, e ele funciona sem qualquer configuração adicional porque seguimos as convenções mencionadas anteriormente. Isso é, temos um model User com uma coluna username e uma password, e usamos um form para postar os dados do usuário para o controller. Essa função retorna se o login foi bem sucedido ou não, e caso ela retorne sucesso, então nós redirecionamos o usuário para a url que configuramos quando adicionamos o AuthComponent em nossa aplicação.

O logout funciona exatamente quando acessamos a url `/users/logout` e irá redirecionar o usuário para a url configurada em logoutUrl anteriormente descrita. Essa url é acionada quando a função `AuthComponent::logout()` obtém sucesso.

Autorização (quem tem permissão de acessar o que)

Como afirmado anteriormente, nós estamos convertendo esse blog em uma ferramenta multi usuário de autoria, e para fazer isso, nós precisamos modificar um pouco a tabela posts para adicionar a referência ao model User:

```
ALTER TABLE posts ADD COLUMN user_id INT(11);
```

Também, é necessária uma pequena mudança no PostsController para guardar a referência do usuário logado para o post criado:

```
// app/Controller/PostsController.php
public function add() {
    if ($this->request->is('post')) {
        $this->request->data['Post']['user_id'] = $this->Auth->user('id'); //␣
↪Adicionada essa linha
        if ($this->Post->save($this->request->data)) {
            $this->Flash->success('Your post has been saved.');
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

A função `user()` fornecida pelo component retorna qualquer coluna do usuário logado no momento. Nós usamos esse metódo para adicionar a informação dentro de request data para que ela seja salva.

Vamos garantir que nossa app evite que alguns autores editem ou apaguem posts de outros. Uma regra básica para nossa aplicação é que usuários admin possam acessar qualquer url, enquanto usuários normais (o papel author) podem somente acessar as actions permitidas. Abra novamente a classe AppController e adicione um pouco mais de opções para as configurações do Auth:

```
// app/Controller/AppController.php

public $components = array(
    'Flash',
    'Auth' => array(
        'loginRedirect' => array('controller' => 'posts', 'action' => 'index'),
        'logoutRedirect' => array('controller' => 'pages', 'action' => 'display',
↪'home'),
        'authorize' => array('Controller') // Adicionamos essa linha
    )
);

public function isAuthorized($user) {
    if (isset($user['role']) && $user['role'] === 'admin') {
        return true; // Admin pode acessar todas actions
    }
    return false; // Os outros usuários não podem
}
```

Nós acabamos de criar um mecanismo de autorização muito simples. Nesse caso os usuários com papel `admin` poderão acessar qualquer url no site quando estiverem logados, mas o restante dos usuários (i.e o papel `author`) não podem acessar qualquer coisa diferente dos usuários não logados.

Isso não é exatamente o que nós queremos, por isso precisamos corrigir nosso metódo `isAuthorized()` para fornecer mais regras. Mas ao invés de fazer isso no AppController, vamos delegar a cada controller para suprir essas regras extras. As regras que adicionaremos para o add de PostsController deve permitir ao autores criarem os posts mas evitar a edição de posts que não sejam deles. Abra o arquivo `PostsController.php` e adicione o seguinte conteúdo:

```
// app/Controller/PostsController.php

public function isAuthorized($user) {
    if (parent::isAuthorized($user)) {
        if ($this->action === 'add') {
            // Todos os usuários registrados podem criar posts
            return true;
        }
        if (in_array($this->action, array('edit', 'delete'))) {
            $postId = (int) $this->request->params['pass'][0];
            return $this->Post->isOwnedBy($postId, $user['id']);
        }
    }
    return false;
}
```

Nós estamos sobreescrevendo a chamada do `isAuthorized()` do AppController e internamente verificando na classe pai se o usuário está autorizado. Caso ele não esteja, permitiremos acesso à ação add, e condicionamente acesso às ações edit e delete. A última coisa que falta implementar, é dizer se usuário é autorizado a editar o post ou não, nós estamos chamando a função `isOwnedBy()` no model Post. Mover lógica para dentro dos models normalmente é uma boa prática. Vamos então implementar essa função:

```
// app/Model/Post.php

public function isOwnedBy($post, $user) {
    return $this->field('id', array('id' => $post, 'user_id' => $user)) === $post;
}
```

Isso conclui então nossa autorização simples e nosso tutorial de autorização. Para garantir o UsersController você pode seguir as mesmas técnicas que usamos para PostsController, você também pode ser mais criativo e codificar algumas coisas mais gerais no AppController para suas próprias regras baseadas em papéis.

Se precisar de mais controle, nós sugerimos que leia o guia completo do Auth /core-libraries/components/authentication seção onde você encontrará mais sobre a configuração do componente, criação de classes de Autorização customizadas, e muito mais.

Sugerimos as seguintes leituras

1. /console-and-shells/code-generation-with-bake Generating basic CRUD code
2. /core-libraries/components/authentication: User registration and login

Aplicação simples controlada por Acl

Nota: Este não é um tutorial para iniciantes. Se você está dando os primeiros passos com o CakePHP nós sugerimos que você explore mais as funcionalidades do framework antes de seguir com esse tutorial.

Neste tutorial você irá criar uma aplicação simples com /core-libraries/components/authentication e /core-libraries/components/access-control-lists. Este tutorial assume que você leu *Blog* e você é familiarizado com /console-and-shells/code-generation-with-bake. Você deve ter alguma experiência com CakePHP, e ser familiarizado com conceitos de MVC. Este tutorial é uma breve introdução à `AuthComponent` e `AclComponent`.

O que você irá precisar

1. Um servidor web funcionando. Nós assumiremos que você está usando o Apache, apesar que as instruções para o uso de outros servidores devem ser bem semelhantes. Existe a possibilidade de alterarmos um pouco a configuração do servidor mas na maioria dos casos o CakePHP irá funcionar sem nenhuma configuração extra.
2. Um servidor de banco de dados. Nós iremos usar o MySQL neste tutorial. O máximo que você precisa saber de SQL é criar uma base de dados: o CakePHP irá assumir as rédeas apartir daí.
3. Conhecimento básico de PHP. Quanto mais programação orientada a objeto você tiver desenvolvido ao longo da vida, melhor: mas não tema se você é um fã de programação procedural.

Iniciando nossa Aplicação

Primeiro, vamos baixar uma nova cópia do CakePHP.

Para baixar uma nova cópia, visite o projeto do CakePHP no GitHub: https://github.com/cakephp/cakephp/tags e baixe a versão estável. Para este tutorial você precisa do último lançamento da versão 2.0.

Você também pode clonar o repositório usando git[52]. `git clone git://github.com/cakephp/cakephp.git`

Uma vez que você baixou uma cópia do CakePHP, configure seu arquivo database.php e altere o valor do Security.salt em app/Config/core.php. Depois disso iremos criar um banco de dados bem simples para construir nossa aplicação. Execute os seguintes comandos SQL em seu banco de dados:

```
CREATE TABLE users (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    password CHAR(40) NOT NULL,
    group_id INT(11) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE groups (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE posts (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id INT(11) NOT NULL,
    title VARCHAR(255) NOT NULL,
    body TEXT,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE widgets (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    part_no VARCHAR(12),
    quantity INT(11)
);
```

[52] http://git-scm.com/

Estas são as tabelas que nós iremos usar para construir o resto de nossa aplicação. Uma vez que nós temos a estrutura de tabelas em nossa base de dados nós podemos começar a «assar» nossa aplicação. Use /console-and-shells/code-generation-with-bake para criar rapidamente seus modelos, controladores e views.

Para usar o bake do CakePHP, execute «cake bake all», isto irá listar as 4 tabelas que você inseriu no MySQL. Selecione «1. Group», e siga os prompts. Repita o proceso para as outras 3 tabelas, e isto irá ter gerado os 4 controladores, modelos e views para você.

Evite o uso de Scaffold neste caso. A criação dos ACOs irá ser seriamente afetada se você gerar os controladores com o uso do Scaffold.

Enquanto estiver gerando os Modelos com o bake, o CakePHP irá automagicamente detectar as associações entre seus Modelos (os relacionamentos entre suas tabelas). Deixe o CakePHP criar corretamente as associações hasMany e belongsTo. Se no prompt você for questionado a escolher hasOne ou hasMany, geralmente você irá precisar de relacionamentos hasMany para este tutorial.

Esqueça as rotas de admin por enquanto, este é um assunto complicado o bastante sem elas. Também esteja certo de **não** adicionar tanto o Acl quanto o Auth Components em nenhum dos seus controladores já que você está gerando eles com o bake. Nós iremos fazer isso em breve. Você agora deve ter modelos, conroladores e views geradas para seus users, groups, posts and widgets.

Adicionando o Auth Component

No momento temos um CRUD funcionando. O bake deveria ter configurado todos os relacionamentos que nós precisamos, se não adicione-os agora. Existem algumas outras etapas que precisam ser concluídas antes de nós adicionarmos o Auth e o Acl Components. Primeiro adicione uma ação login e logout em `UsersController`:

```
public function login() {
    if ($this->request->is('post')) {
        if ($this->Auth->login()) {
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Session->setFlash(__('Your username or password was incorrect.'));
    }
}

public function logout() {
    //Leave empty for now.
}
```

Depois crie a seguinte view `app/View/Users/login.ctp`:

```
echo $this->Form->create('User', array('action' => 'login'));
echo $this->Form->inputs(array(
    'legend' => __('Login'),
    'username',
    'password'
));
echo $this->Form->end('Login');
```

Agora nós iremos atualizar nosso modelo de User para ele fazer um hash nas senhas antes que elas sejam gravadas no banco de dados. Gravar senhas sem encriptação é extremamente inseguro e o AuthComponent espera que suas senhas estejam encriptadas. Em `app/Model/User.php` adicione o seguinte:

```
App::uses('AuthComponent', 'Controller/Component');
class User extends AppModel {
    // other code.
```

```
    public function beforeSave($options = array()) {
        $this->data['User']['password'] = AuthComponent::password(
          $this->data['User']['password']
        );
        return true;
    }
}
```

Agora nós precisamos fazer algumas modificações em `AppController`. Se você não possui `/app/Controller/AppController.php`, crie o arquivo. Como queremos que toda a nossa aplicação seja controlada por Auth e Acl, nós iremos configurá-los em `AppController`:

```
class AppController extends Controller {
    public $components = array(
        'Acl',
        'Auth' => array(
            'authorize' => array(
                'Actions' => array('actionPath' => 'controllers')
            )
        ),
        'Session'
    );
    public $helpers = array('Html', 'Form', 'Session');

    public function beforeFilter() {
        //Configure AuthComponent
        $this->Auth->loginAction = array(
          'controller' => 'users',
          'action' => 'login'
        );
        $this->Auth->logoutRedirect = array(
          'controller' => 'users',
          'action' => 'login'
        );
        $this->Auth->loginRedirect = array(
          'controller' => 'posts',
          'action' => 'add'
        );
    }
}
```

Antes de configurar o ACL completamente nós precisamos de alguns users e groups adicionados. Com o `AuthComponent` em uso nós não vamos conseguir acessar nenhuma de nossas ações já que não estamos logados. Para resolver isso vamos adicionar algumas excessões em `AuthComponent` para ele permitir que criemos alguns groups e users. Em **ambos** arquivos `GroupsController` e `UsersController` adicione o seguinte:

```
public function beforeFilter() {
    parent::beforeFilter();

    // For CakePHP 2.0
    $this->Auth->allow('*');

    // For CakePHP 2.1 and up
    $this->Auth->allow();
}
```

As linhas acima dizem para o AuthComponent permitir acesso público para todas as ações. Isto é temporário e irá ser

removido uma vez que nós cadastramos alguns users e groups em nossa base de dados. Mas não adicione nenhum user ou group ainda.

Criando as tabelas Db Acl

Antes de criarmos qualquer users ou groups seria interessante conecta-los ao Acl. Mas nós ainda não temos nenhuma tabela do Acl e se você tentar acessar qualquer página, você irá obter um erro de tabela inexistente («Error: Database table acos for model Aco was not found.»). Para remover estes erros nós precisamos rodar um schema. Em um terminal rode o seguinte:

```
./Console/cake schema create DbAcl
```

Este schema irá perguntar se você quer apagar e criar as tabelas. Diga sim para deletar e criar as tabelas.

Se você não tiver acesso ao terminal, ou se estiver tendo problema ao usar o console, você pode rodar o arquivo sql situado em /path/to/app/Config/Schema/db_acl.sql.

Com os controladores prontos para gravar dados e as tabelas Acl inicializadas nós estamos prontos para seguir adiante, correto? Não necessariamente, ainda temos um pouco de trabalho para fazer nos modelos de User e Group. Mais precisamente, fazer com que eles auto-magicamente se conectem ao Acl.

Funcionando como um Requester

Para o Auth e o Acl funcionarem corretamente nós precisamos relacionar os users e groups com registros nas tabelas do ACL. Para conseguirmos isso, iremos usar o `AclBehavior`. O `AclBehavior` permite que automagicamente conectemos nossos modelos com as tabelas do Acl. O seu uso requer a implementação do `parentNode()` em seu modelo. Em nosso modelo de `User` iremos adicionar o seguinte:

```
class User extends AppModel {
    public $belongsTo = array('Group');
    public $actsAs = array('Acl' => array('type' => 'requester'));

    public function parentNode() {
        if (!$this->id && empty($this->data)) {
            return null;
        }
        if (isset($this->data['User']['group_id'])) {
            $groupId = $this->data['User']['group_id'];
        } else {
            $groupId = $this->field('group_id');
        }
        if (!$groupId) {
            return null;
        }
        return array('Group' => array('id' => $groupId));
    }
}
```

Depois em nosso modelo de `Group` adicionamos o seguinte:

```
class Group extends AppModel {
    public $actsAs = array('Acl' => array('type' => 'requester'));

    public function parentNode() {
        return null;
```

```
    }
}
```

O que isto faz é conectar os modelos de `Group` e `User` ao Acl, e dizer ao CakePHP que toda vez que você criar um User ou um Group você também quer um registro na tabela `aros`. Isto faz o gerenciamento do Acl extremamente simples já que seu AROs se torna visivelmente conectado as suas tabelas `users` and `groups`. Cada vez que você ou deleta um usuário/grupo a tabela Aro é atualizada.

Nossos controladores e modelos estão agora preparados para adicionar algum dado, e nossos modelos de `Group` and `User` estão atrelados à tabela do ACL. Adicione alguns groups e users usando os formulários gerados pelo bake acessando http://example.com/groups/add e http://example.com/users/add. Eu criei os seguintes groups:

- administrators
- managers
- users

Eu também criei um user em cada group para então ter um usuário de cada grupo de acesso para testar futuramente. Anote tudo em algum lugar ou use senhas fáceis para não esquecer. Se você executar `SELECT * FROM aros;` em um prompt mysql ele deve retornar algo parecido com o seguinte:

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    4 |
|  2 |      NULL | Group |           2 | NULL  |    5 |    8 |
|  3 |      NULL | Group |           3 | NULL  |    9 |   12 |
|  4 |         1 | User  |           1 | NULL  |    2 |    3 |
|  5 |         2 | User  |           2 | NULL  |    6 |    7 |
|  6 |         3 | User  |           3 | NULL  |   10 |   11 |
+----+-----------+-------+-------------+-------+------+------+
6 rows in set (0.00 sec)
```

Isto nos mostra que temos 3 groups e 3 users. Os users são aninhados dentro de groups, isto quer dizer que podemos definir permissões por grupo ou por usuário.

Permissões por Grupo

Se você deseja criar permissões por grupo, nós precisamos implementar `bindNode()` no modelo de `User`:

```
public function bindNode($user) {
    return array('model' => 'Group', 'foreign_key' => $user['User']['group_id']);
}
```

Depois modifique o `actsAs` do modelo de `User` e desabilite a opção enabled:

```
public $actsAs = array('Acl' => array('type' => 'requester', 'enabled' => false));
```

Estas duas mudanças irão dizer ao ACL para evitar a checagem no ARO de `User` para checar somente o ARO de `Group`. Isto também evita o afterSave de ser chamado.

Observação: Todo user precisa ter um `group_id` relacionado para isso funcionar.

Agora a tabela `aros` irá parecer com isto:

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
```

```
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    2 |
|  2 |      NULL | Group |           2 | NULL  |    3 |    4 |
|  3 |      NULL | Group |           3 | NULL  |    5 |    6 |
+----+-----------+-------+-------------+-------+------+------+
3 rows in set (0.00 sec)
```

Creating ACOs (Access Control Objects)

Agora que temos nossos users and groups(aros), nós podemos começar a inserir nossos controladores dentro do Acl e a setar permissões para nossos groups e users, como também habilitar login / logout.

Nossos AROs estão sendo criados automaticamente quando novos users e groups são criados. Que tal uma forma de auto-gerar os ACOs dos nossos controladores e suas ações? É... infelizmente não há uma forma mágica no core do CakePHP para realizar isto. Mas as classes do core oferecem alguns meios para criar manualmente os ACOs. Você pode criar os objetos do ACO apartir do terminal do Acl ou você pode usar o `AclComponent`. Criar Acos usando o terminal pode ser algo como:

```
./Console/cake acl create aco root controllers
```

Enquanto usar o AclComponent irá ser algo como:

```
$this->Acl->Aco->create(array('parent_id' => null, 'alias' => 'controllers'));
$this->Acl->Aco->save();
```

Ambos os exemplos iriam criar a «raíz»(root) ou o nível mais alto do ACO que irá se chamar "controllers". A razão desse nódulo raiz é facilitar o allow/deny num escopo global da aplicação e permitir o uso do Acl para questões não relacionadas à controladores/ações por exemplo a checagem de permissão de gravação de modelos. Como estamos usando um ACO raíz nós precisamos fazer uma pequena modificação na configuração do nosso `AuthComponent`. O `AuthComponent` precisa ser informado sobre a existência deste nódulo raíz para que então quando ele for realizar a checagem do ACL ele use o caminho correto quando for procurar por controladores/ações. Dentro de `AppController` certifique-se de que o array `$components` contem a chave `actionPath` definida:

```
class AppController extends Controller {
    public $components = array(
        'Acl',
        'Auth' => array(
            'authorize' => array(
                'Actions' => array('actionPath' => 'controllers')
            )
        ),
        'Session'
    );
```

Para continuar o tutorial acesse *Simple Acl controlled Application - part 2*.

Simple Acl controlled Application - part 2

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[53] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

[53] https://github.com/cakephp/docs

CAPÍTULO 10

Apêndices

Os apêndices contêm informações sobre os novos recursos introduzidos na versão 2.0 e o guia de migração da versão 1.3 para a versão 2.0.

Guia de Migração para a Versão 2.10

2.10 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[54] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.9

2.9 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[55] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[54] https://github.com/cakephp/docs
[55] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.8

2.8 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[56] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.7

2.7 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[57] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.6

2.6 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[58] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

[56] https://github.com/cakephp/docs
[57] https://github.com/cakephp/docs
[58] https://github.com/cakephp/docs

Guia de Migração para a Versão 2.5

2.5 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[59] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.4

2.4 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[60] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.3

2.3 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[61] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.2

2.2 Migration Guide

[59] https://github.com/cakephp/docs
[60] https://github.com/cakephp/docs
[61] https://github.com/cakephp/docs

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[62] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.1

Guia de Migração para a Versão 2.1

O CakePHP 2.1 é um atualização totalmente compatível com API do 2.0. Esta página apresenta as mudanças e melhorias feitas no 2.1.

AppController, AppHelper, AppModel and AppShell

Como essas classes foram removidas do núcleo do CakePHP, agora elas são obrigatórias em sua aplicação. Se você não tiver essas classes, você pode usar o seguinte durante a atualização:

```
// app/View/Helper/AppHelper.php
App::uses('Helper', 'View');
class AppHelper extends Helper {
}

// app/Model/AppModel.php
App::uses('Model', 'Model');
class AppModel extends Model {
}

// app/Controller/AppController.php
App::uses('Controller', 'Controller');
class AppController extends Controller {
}

// app/Console/Command/AppShell.php
App::uses('Shell', 'Console');
class AppShell extends Shell {
}
```

Se sua aplicação já tiver esses arquivos/classes você não precisa fazer nada. Além disso, se você estiver utilizando o PagesController do núcleo do CakePHP, você precisa copiá-lo para a pasta app/Controller também.

Arquivos .htaccess

Os arquivos `.htaccess` foram alterados, você deve lembrar de atulizá-los ou a atualizar o esquema de reescrita de `URL re-writing` para combinar com as atualizações feitas.

[62] https://github.com/cakephp/docs

Models

- O callback‘‘beforeDelete‘‘ será disparado antes dos callbacks beforeDelete dos behaviors. Isto o torna consistente com o resto dos eventos disparados na camada de modelo.
- O método `Model::find('threaded')` agora aceita o parâmetro `$options['parent']` se estiver usando outro campo como `parent_id`. Além disso, se o model tem o TreeBehavior e configurado com outro campo definido para o pai, o `threaded` irá encontrá-lo e o utilizar por padrão.
- Parâmetros para consultas usando instruções preparadas (`prepared statements`) agora será parte do dump SQL.
- Arrays de validação agora podem ser mais específicos quando um campo é obrigatório. A chave `required` agora aceita `create` e `update`. Esses valores farão o campo obrigatório quando criar ou atualizar.

Behaviors

TranslateBehavior

- `I18nModel` foi movida para um arquivo separado.

Exceções

A renderização padrão de exceções agora inclui os *stack traces* mais detalhados, incluindo trechos de arquivos e os argumentos para todas as funções.

Utilidade

Debugger

- `Debugger::getType()` foi adicionada. Pode ser utilizada para obter o tipo das variáveis.
- `Debugger::exportVar()` foi modificada para criar uma saída mais legível e útil.

debug()

debug() agora utiliza `Debugger` internamente. Isto o torna mais consistente com o Debugger, e tira proveito das melhorias feitas lá.

Set

- `Set::nest()` foi adicionada. Recebe uma matriz simples e retorna uma matriz aninhada.

File

- `File::info()` inclui o tamanho do arquivo e o seu mimetype.
- `File::mime()` foi adicionada.

Cache

- `CacheEngine` foi movida para um arquivo separado.

Configure

- `ConfigReaderInterface` foi movida para um arquivo separado.

App

- `App::build()` agora tem a capacidade de registrar novos pacotes usando `App::REGISTER`. Veja app-build-register para mais informações.
- As classes que não podem ser encontradas nos caminhos configurados serão pesquisados dentro de `APP`, como um caminho alternativo. Isso torna o carregamento automático dos diretórios aninhados em `app/Vendedor` mais fácil.

Console

Test Shell

Um novo TestShell foi adicionado. Ele reduz a digitação necessária para executar os testes unitários, e oferece uma interface baseada nos caminhos dos arquivos:

```
# Run the post model tests
Console/cake test app/Model/Post.php
Console/cake test app/Controller/PostsController.php
```

O antigo shell testsuite e sua sintaxe ainda estão disponíveis.

General

- Arquivos gerados não contém timestamps com o dia/hora da geração.

Rotas

Router

- As rotas agora podem usar uma sintaxe especial `/**` para incluir todos os argumentos finais como um único argumento passado. Veja a seção sobre *Connecting Routes* para mais informações.
- `Router::resourceMap()` foi adicionada.
- `Router::defaultRouteClass()` foi adicionada. Este método permite que você defina a classe padrão usada para todas as rotas definidas.

Network

CakeRequest

- Adicionado `is('requested')` e `isRequested()` para a detecção de `requestAction`.

CakeResponse

- Adicionado `CakeResponse::cookie()` para a configuração de *cookies*.
- Foi adicionada uma série de métodos para *Fine tuning HTTP cache*

Controller

Controller

- O *`Controller::$uses`* foi modificado, seu valor padrão agora é `true` em vez de `false`. Além disso, valores diferentes são tratados de maneira ligeiramente diferente, mas irá comportar o mesmo na maioria dos casos.
  - `true` Irá carregar o modelo padrão e mesclar com AppController.
  - Um array irá carregar os modelos e mesclar com AppController.
  - An empty array will not load any models other than those declared in the base class.
  - Um array vazio não vai carregar outros modelos que não os declarados na classe base.
  - `false` não irá carregar qualquer modelo, e não vai se fundir com a classe base também.

Componentes

AuthComponent

- `AuthComponent::allow()` não aceita mais `allow('*')` como um curinga para todas as ações. Basta usar `allow()`. Isso unifica a API entre allow() e deny().
- A opção `recursive` foi adicionada a todos os adaptadores de autenticação. Permite controlar mais facilmente as associações armazenados na sessão.

AclComponent

- `AclComponent` não mais inflexiona o nome da classe usada para `Acl.classname`. Em vez disso utiliza o valor como é fornecido.
- Implementações do Acl agora devem ser colocadas em `Controller/Component/Acl`.
- Implementações do Acl agora devem ser movidas da pasta `Component` para a pasta `Component/Acl`. Por exemplo: se sua classe Acl se chama `CustomAclComponent`, e está em `Controller/Component/CustomAclComponent.php`. Ela deve ser movida para `Controller/Component/Acl/CustomAcl.php` e renomeada para `CustomAcl`.
- `DbAcl` foi movida para um arquivo separado.
- `IniAcl` foi movida para um arquivo separado.

- `AclInterface` foi movida para um arquivo separado.

Helpers

TextHelper

- `TextHelper::autoLink()`, `TextHelper::autoLinkUrls()`, `TextHelper::autoLinkEmails()` escapa o HTML por padrão. Você pode controlar este comportamento com a opção `escape`.

HtmlHelper

- `HtmlHelper::script()` teve a opção `block` adicionada.
- `HtmlHelper::scriptBlock()` teve a opção `block` adicionada.
- `HtmlHelper::css()` teve a opção `block` adicionada.
- `HtmlHelper::meta()` teve a opção `block` adicionada.
- O parâmetro `$startText` do `HtmlHelper::getCrumbs()` pode ser um array. Isto dá mais controle e flexibilidade.
- `HtmlHelper::docType()` o padrão agora é o html5
- `HtmlHelper::image()` agora tem a opção `fullBase`.
- `HtmlHelper::media()` foi adicionado. Você pode usar este método para criar elementos de vídeo/audio do HTML5.
- O suporte a *plugin syntax* foi adicionado nos métodos `HtmlHelper::script()`, `HtmlHelper::css()`, `HtmlHelper::image()`. Agora você pode facilmente vincular recursos de plugins usando `Plugin.asset`.
- `HtmlHelper::getCrumbList()` teve o parâmetro `$startText` adicionado.

View

- *`View::$output`* está obsoleto.
- `$content_for_layout` está obsoleto. Use `$this->fetch('content');` instead.
- `$scripts_for_layout` está obsoleto. Use o seguinte:

```
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
```

  `$scripts_for_layout` ainda está disponível, mas a API *view blocks* API é mais flexível e extensível.

- A sintaxe `Plugin.view` está agora disponível em todos os lugares. Você pode usar esta sintaxe em qualquer lugar que você fizer referência ao nome de uma *view*, *layout* ou *element*.
- A opção `$options['plugin']` para *`element()`* está obsoleta. Em vez disso você deve utilizar `Plugin.element_name`.

Content type views

Duas classes de exibição foram adicionadas ao CakePHP. A `JsonView` e a `XmlView` permite gerar facilmente views XML e JSON. Você pode aprender mais sobre essas classes na seção *JSON and XML views*.

Estendendo as views

`View` has a new method allowing you to wrap or "extend" a view/element/layout with another file. See the section on *Estendendo Views* for more information on this feature.

Temas

A classe `ThemeView` está obsoleta em favor da classe `View`. Simplesmente defina o `$this->theme = 'MyTheme` que o suporte a temas será habilitado, e todas as classes de View personalizadas que estendem da `ThemeView` deve estender de `View`.

Blocos de View

Blocos de View são uma maneira flexível de criar slots ou blocos em suas views. Os blocos substituem `$scripts_for_layout` com uma API mais robusta e flexível. Consulte a seção sobre *Usando Blocos de Views (Visões)* para mais informações.

Helpers

Novos callbacks

Dois novos callbacks foram adicionados aos Helpers. `Helper::beforeRenderFile()` e `Helper::afterRenderFile()` esses novos callbacks são disparados antes/depois que cada fragmento da view é renderizado. Isto inclui elements, layouts e views.

CacheHelper

- As tags `<!--nocache-->` agora funcionam corretamente dentro dos elementos.

FormHelper

- O FormHelper agora omite campos desabilitados a partir do hash dos campos protegidos. Isso torna o trabalho com `SecurityComponent` e os inputs desabilitados mais fácil.
- A opção `between` quando utilizado em conjunto com os radio inputs, agora se comporta de forma diferente. O valor do `between` agora é colocado entre a legenda e o primeiro input.
- A opção `hiddenField` dos campos checkbox pode agora ser definida para um valor específico, como "N" ao invés de apenas 0.
- O atributo `for` para campos datetime agora reflete o primeiro campo gerado. Isso pode resultar na mudança do atributo `for` de acordo com os campo geradas.
- O atributo `type` para `FormHelper::button()` pode ser removido agora. O padrão ainda é "submit".

- `FormHelper::radio()` agora permite que você desabilite todas as opções. Você pode fazer isso definindo `'disabled' => true` ou `'disabled' => 'disabled'` no array `$attributes`.

PaginatorHelper

- `PaginatorHelper::numbers()` agora possui a opção `currentClass`.

Testando

- Web test runner agora exibe a versão do PHPUnit.
- Web test runner agora mostra os testes da aplicação por padrão.
- Fixtures podem ser criados em datasources que não seja $test.
- Modelos carregados usando o `ClassRegistry` e usando outro datasource vai ter o nome de seu datasource prefixado com `test_` (por exemplo, o datasource *master* irá tentar usar `test_master` no testsuite)
- Os casos de teste são gerados com os métodos de configuração específicos.

Eventos

- Um novo sistema de eventos genérico foi construído e que substituiu a forma como callbacks são disparados. Isso não deve representar qualquer alteração em seu código.
- Você pode enviar seus próprios eventos e callbacks para serem anexados, útil para a comunicação entre plugins e fácil desacoplamento de suas classes.

New Features in CakePHP 2.1

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[63] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Guia de Migração para a Versão 2.0

2.0 Migration Guide

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[64] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[63] https://github.com/cakephp/docs
[64] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

New Features in CakePHP 2.0

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[65] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

PHPUnit Migration Hints

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[66] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Migrando da Versão 1.2 para 1.3

Migrating from CakePHP 1.2 to 1.3

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[67] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

New features in CakePHP 1.3

Nota: A documentação não é atualmente suportada pela lingua portuguesa nesta página.

Por favor, sinta-se a vontade para nos enviar um pull request no Github[68] ou use o botão **Improve This Doc** para propor suas mudanças diretamente.

[65] https://github.com/cakephp/docs
[66] https://github.com/cakephp/docs
[67] https://github.com/cakephp/docs
[68] https://github.com/cakephp/docs

Você pode referenciar-se à versão inglesa no menu de seleção superior para obter informações sobre o tópico desta página.

Informações Gerais

Glossary

routing array An array of attributes that are passed to `Router::url()`. They typically look like:

```
array('controller' => 'posts', 'action' => 'view', 5)
```

HTML attributes An array of key => values that are composed into HTML attributes. For example:

```
// Given
array('class' => 'my-class', 'target' => '_blank')

// Would generate
class="my-class" target="_blank"
```

If an option can be minimized or accepts it's name as the value, then `true` can be used:

```
// Given
array('checked' => true)

// Would generate
checked="checked"
```

plugin syntax Plugin syntax refers to the dot separated class name indicating classes are part of a plugin. E.g. `DebugKit.Toolbar` The plugin is DebugKit, and the class name is Toolbar.

dot notation Dot notation defines an array path, by separating nested levels with `.` For example:

```
Asset.filter.css
```

Would point to the following value:

```
array(
    'Asset' => array(
        'filter' => array(
            'css' => 'got me'
        )
    )
)
```

CSRF Cross Site Request Forgery. Prevents replay attacks, double submissions and forged requests from other domains.

routes.php A file in APP/Config that contains routing configuration. This file is included before each request is processed. It should connect all the routes your application needs so requests can be routed to the correct controller + action.

DRY Don't repeat yourself. Is a principle of software development aimed at reducing repetition of information of all kinds. In CakePHP DRY is used to allow you to code things once and re-use them across your application.

CAPÍTULO 11

Indices and tables

- genindex

Índice